MASTER EN SCIENCES INFORMATIQUES

Implémentation de prototypes de widgets pour la navigation au stylet dans des scènes 2D et 3D

Vo, Minh Thu

Award date:
2008

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'informatique

Année académique 2007 – 2008

Implémentation de prototypes de widgets pour la navigation au stylet dans des scènes 2D et 3D

Võ Minh Thu

Mémoire présenté en vue de l'obtention du grade de maître en informatique

Résumé

Le Bureau Virtuel et le logiciel EsQUIsE ont été développés par le Lucid Group pour fournir un environnement familier à l'architecte et supporter le processus de conception. Dans ce travail, nous avons pour objectif de créer des prototypes de widgets pour la navigation dans des scènes en deux et en trois dimensions. Nous exploitons des techniques d'interactions spécifiques afin de permettre un usage adapté au Bureau Virtuel et au concepteur. Pour faciliter le développement rapide d'applications interactives, nous développons une bibliothèque de programmation sur le principe d'une évaluation dataflow synchrone. Cette bibliothèque permet de réagir à l'absence d'évènement au cours d'un instant.

Mots clés – techniques d'interactions, contrôle de caméra, navigation 3D, manipulation directe, programmation réactive fonctionnelle.

Abstract

The Virtual Desktop and the EsQUIsE software were developed by the Lucid Group to provide a familiar environment to the architect and support the design process. In this work, we aim to create widgets prototypes for the navigation in two and three dimensions scenes. We exploit specific interaction techniques in order to create a use adapted to the Virtual Desktop and the designer. To make the rapid interactive applications development easier, we build a programming library based on a synchronous dataflow evaluation. With this library, it is possible to react to an event absence during an instant.

Keywords – interaction techniques, camera controls, 3D navigation, direct manipulation, functional reactive programming.

Avant-propos

Ce mémoire est principalement le résultat d'un stage de quatre mois effectué au sein de l'équipe du Lucid Group, un laboratoire de recherche du département ArGEnCo (Architecture, Géologie, Environnement & Constructions) de la Faculté des Sciences Appliquées de l'Université de Liège. Le stage s'est déroulé sous la supervision de Monsieur Pierre Leclercq et a consisté en l'implémentation de prototypes de widgets. Ces widgets sont destinés à permettre la navigation dans des scènes virtuelles 2D et 3D pour les applications menées dans le laboratoire, en particulier les logiciels EsQUIsE et SketSha.

Je tiens à remercier mon promoteur, Madame Monique Noirhomme, pour son aide précieuse ainsi que pour son soutien pendant la réalisation de ce mémoire. Je remercie aussi Monsieur Pierre Leclercq pour sa supervision et ses conseils lors du stage. J'adresse également mes remerciements à l'équipe du Lucid Group pour son accueil : Roland Juchmes, Marine Maréchal, Stéphane Safin, Anne Detheux, Jean-François Vandamme, Vincent Delfosse, Dimitri Schmitz, Catherine Elsen, Jean-Noël Demaret, Arnaud Dawans et Rachelle Vafidis.

Table des matières

Table des figures

Glossaire

API

Application Programming Interface. Il s'agit des fonctionnalités exposées par une bibliothèque de programmation.

CAO

Conception assistée par ordinateur.

Dataflow

Principe où le changement de valeur d'une variable force le recalcul des variables qui en dépendent.

Panning

Translation du point de vue parallèlement à l'écran.

Widget

Elément d'interface graphique. Par exemple, un menu déroulant ou une boîte de dialogue.

Introduction

En architecture, les outils informatiques traditionnels imposent une rigueur de fonctionnement incompatible avec le travail de conception. Le concepteur-architecte préfère généralement l'utilisation du papier et du crayon qui laissent place aux idées, encore floues au début, et à la recherche graphique. Il peut ainsi créer rapidement, explorer et *construire* une solution plutôt que d'en définir une version finale, dans le système de CAO.

Afin d'offrir les avantages de l'outil informatique tout en laissant le concepteur s'exprimer de sa façon habituelle, le Lucid Group a élaboré le Bureau Virtuel et le logiciel EsQUIsE. Dans cet environnement, le concepteur peut produire des esquisses à main levée et le système se charge d'inférer leur signification. L'architecte n'a pas à utiliser des boîtes de dialogue pour communiquer ses intentions mais il peut s'exprimer comme s'il partageait un croquis avec un collègue.

EsQUIsE peut également, à partir des croquis de bâtiments, produire un modèle en trois dimensions que l'architecte peut inspecter. Que ce soit pour visualiser le plan en deux dimensions ou pour regarder le modèle en trois dimensions, les créateurs du Bureau Virtuel et d'EsQUIsE désirent proposer à l'utilisateur un moyen de contrôler le point de vue qui réponde aux motivations premières. Le point de vue doit pouvoir être manipulé simplement au stylet et son contrôle doit se laisser oublier du concepteur pour qu'il puisse se concentrer sur son travail.

Dans ce mémoire, nous présentons un éventail de possibilités susceptibles de répondre à ces exigences, sous forme de différents *widgets* (des éléments d'interface graphique). Le premier chapitre décrit le contexte dans lequel le stage s'est déroulé. Il présente les logiciels EsQUIsE et SketSha auxquels la conception de la navigation dans des scènes 2D et 3D est liée. Il est également l'occasion d'indiquer les limites des interactions possibles entre l'utilisateur et l'interface de navigation.

Le deuxième chapitre introduit les concepts de base de la navigation dans une scène virtuelle 2D ou 3D, à savoir le contrôle d'une caméra virtuelle placée dans la scène. Ces concepts sont exploités dans la suite du mémoire.

Le chapitre 3 expose la manière dont la navigation dans des logiciels existants est permise ainsi qu'un état de l'art en matière de techniques d'interactions qui sont adaptées à l'utilisation d'un stylet et du Bureau Virtuel. C'est à partir de ces informations que nous avons choisi avec l'équipe du Lucid Group le style et les principes des différents widgets.

Le chapitre 4 expose d'une part les bases de la programmation 3D telle qu'elle se fait au Lucid Group et d'autre part les différentes implémentations de widgets.

Nous avons fait tester les widgets par des membres de l'équipe et des étudiants en architecture. Les observations dégagées sont précisées dans le chapitre 5.

Nous avons trouvé intéressant de développer une bibliothèque de programmation que nous détaillons dans le chapitre 6. Elle permet la réalisation rapide de logiciels interactifs et notamment des widgets comme ceux des chapitres qui précèdent.

Enfin, nous développons nos conclusions et suggérons un certain nombre de possibilités qui nous ont parues intéressantes à explorer.

Chapitre 1

Contexte

Ce travail concerne le développement de *widgets* (éléments d'interface graphique) pour la navigation dans des applications 2D et 3D. Ces applications sont réalisées au Lucid Group, un laboratoire de recherche du département ArGEnCo (Architecture, Géologie, Environnement & Constructions) de l'Université de Liège.

Par navigation, nous entendons le déplacement par l'utilisateur du point de vue adopté pour visualiser la scène virtuelle gérée par l'application. La scène d'une application 2D représentera par exemple une feuille de papier sur laquelle l'utilisateur dessine le plan d'un bâtiment. Une scène 3D pourrait être ce même bâtiment une fois mis en volume.

Les applications existantes sur le marché fournissent bien entendu la possibilité de naviguer. Dans ce travail, le besoin de nouvelles façons de naviguer est dû à la spécificité des applications développées au Lucid Group, en particulier EsQUIsE et SketSha. Celles-ci doivent permettre aux architectes de débuter l'étape de conception directement de manière informatisée.

Lors de la phase de conception, l'architecte a l'habitude de travailler par croquis sur papier. Ce mode de travail ne possède pas la rigueur des outils de CAO traditionnels et convient mieux aux idées naissantes du travail à réaliser.

La programmation des prototypes de widgets s'est déroulée au sein du Lucid Group et le code écrit était intégré aux développements du laboratoire.

1.1 EsQUIsE

En architecture, la conception débute généralement sur papier et le passage à l'outil informatique n'est effectué que plus tard. La rigueur que les outils de CAO traditionnels imposent au concepteur entrave la liberté nécessaire à cette étape. L'architecte doit pouvoir se concentrer sur son travail plutôt que sur les détails d'utilisation d'un logiciel. De plus, le crayon et la table à dessin restent les outils privilégiés [SBL05].

Afin de libérer le concepteur des contraintes habituelles des outils informatiques, le logiciel EsQUIsE [JL04] fonctionne sur un Bureau Virtuel et propose une interface d'utilisation aussi proche que possible de celle du crayon sur le papier.

1.1.1 Atouts

Le Bureau Virtuel se compose d'un tableau blanc digital de la taille d'une table à dessin et de deux projecteurs suspendus au plafond (fig. 1.1). Deux projecteurs sont nécessaires pour assurer une définition d'image suffisante en couvrant toute la longueur du tableau. Le tableau est placé horizontalement et forme une table. Avec un stylet semblable à un simple stylo, il sert de périphérique d'entrée semblable à un très grand écran tactile.

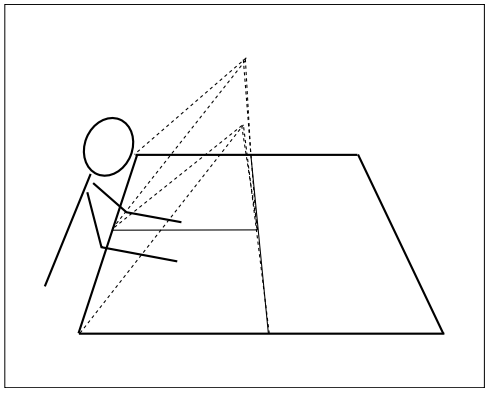


FIG. 1.1 – *Le Bureau Virtuel se compose d'un tableau blanc digital placé horizontalement et de deux projecteurs placés pour limiter l'ombre de l'utilisateur.*

EsQUIsE permet de réaliser des croquis à main levée à partir desquels une maquette en trois dimensions peut être générée. EsQUIsE est capable d'interpréter, en plus du croquis lui-même, des annotations (comme la hauteur d'un mur) ou des symboles (par exemple le symbole d'un escalier). EsQUIsE est construit autour d'un système multi-agents qui analysent les traits des dessins à mesure qu'ils sont formés. Ces

agents ont pour tâche d'inférer des informations liées au domaine de l'architecture sans que l'utilisateur n'ait besoin de les expliciter. Le modèle interne d'EsQUIsE ainsi obtenu sert de base à d'autres composants du logiciel (appelés évaluateurs). Ces composants réalisent des calculs habituellement reportés après la conception. Par exemple, un évaluateur peut calculer la performance thermique d'un bâtiment. Grâce aux résultats de ces calculs, le concepteur peut faire évoluer son projet au fur et à mesure. Autrement, il n'aurait pu procéder à des modifications qu'après une mise en forme très aboutie.

Le concept de croquis et d'interaction "au crayon" ne concerne pas uniquement l'entrée initiale des données. Lorsque EsQUIsE crée une maquette 3D à partir d'un croquis, il est à noter que le dessin tracé par l'architecte reste en l'état. Notamment, les lignes qui sont identifiées par le logiciel comme des segments de droite (par exemple des murs) ne sont pas remplacées par des segments de droite à l'affichage et conservent leur apparence de croquis.

1.1.2 Limitations

A l'heure actuelle, l'intérêt de l'affichage d'un modèle 3D est malheureusement un peu diminué par la pauvreté des possibilités de navigation qu'offre EsQUIsE, tant au niveau des opérations de navigation proposées qu'au niveau de la sélection de l'opération et de son utilisation.

Rotation

La rotation du modèle 3D se fait uniquement autour d'un point fixe. Lorsque seule une partie de la scène est visible et que le centre de rotation est en dehors de l'écran, effectuer une rotation déplace ce qui était visible[1].

Translation

Les opérations de *panning* (déplacement de la vue dans un plan parallèle à l'écran) et de zoom sont moins dépendantes d'un point précis mais leur utilisation, comme pour la rotation, n'est pas satisfaisante. A chaque opération correspond un mode et

[1] Imaginez être assis au bout d'une table assez longue. Posée devant vous se trouve une assiette que vous pouvez faire tourner. Si le centre de rotation se trouve au centre de l'assiette, celle-ci restera devant vous. Par contre, si le centre de rotation se trouve à l'autre bout de la table, l'assiette s'éloignera rapidement de vous. La situation est d'autant plus gênante si vous ne pouvez voir que l'emplacement initial de l'assiette plutôt que tout l'espace autour de la table.

l'application ne peut être que dans un seul mode à la fois. Utiliser une opération de navigation se fait par l'intermédiaire d'un menu qui comporte un bouton pour chaque mode. Choisir une opération se fait en cliquant sur un bouton qui fixe le mode courant (rotation, panning ou zoom).

Menu classique et modes

Le principe d'un menu traditionnel n'est pas acceptable en raison de la grande surface d'affichage utilisée qui impose des aller-retour entre le lieu où se trouve le menu et le lieu d'utilisation du mode courant. Pour le responsable du développement d'EsQUIsE, le principe même d'activer et de désactiver un mode (éventuellement pour revenir dans le mode de base, lorsqu'aucun bouton n'est enfoncé) doit être évité. De ce fait, il est demandé de trouver d'autres moyens de passer d'une opération de navigation à une autre. Ces moyens doivent mieux supporter le but initial d'EsQUIsE : libérer au maximum l'architecte des contraintes de l'utilisation de l'outil informatique tout en maintenant ses bénéfices.

1.2 SketSha

EsQUIsE est écrit dans le langage de programmation Common Lisp. Pour faciliter des développements futurs, notamment lorsqu'ils se font avec d'autres groupes de programmeurs, le Lucid Group a décidé de réécrire EsQUIsE sous une forme plus modulaire, en suivant une programmation orientée objet et en utilisant le langage Java, beaucoup plus populaire que Common Lisp. La réécriture d'EsQUIsE, appelée SketSha, a commencé par un programme de dessin collaboratif utilisé sur le Bureau Virtuel. L'ajout du système multi-agents et des autres modules doit se faire plus tard. Il s'agit donc pour l'heure d'une application 2D. Pour cette application, comme avec EsQUIsE, le Lucid Group désirait expérimenter d'autres manières de naviguer que le simple menu. Les opérations de navigation sont plus simples et moins nombreuses que dans le cas d'une scène 3D. Néanmoins, l'utilisation sur le Bureau Virtuel de SketSha en fait un bon terrain pour tester de nouvelles façons de changer de mode.

1.3 La conception avec le Bureau Virtuel

Le Bureau Virtuel et EsQUIsE ont été créés pour permettre à l'informatique de répondre de façon très spécifique et optimale aux besoins du concepteur-architecte. En effet, pendant l'étape de conception, l'architecte préfère habituellement travailler

avec les outils papier et crayon. Ceux-ci le laisse exprimer, explorer et construire progressivement une solution. Contrairement à une application de CAO, les traits sur le papier n'ont pas besoin d'être rectilignes ou composés avec précision. L'esquisse fait partie d'un processus où il est bénéfique de pouvoir rapidement exprimer une idée, même floue ou abstraite et où l'ambiguïté des traits participe au cheminement créatif.

Avec le support papier, le concepteur dispose de beaucoup de liberté, il peut le positionner devant lui comme il l'entend. Il peut aussi se déplacer autour de la table sur laquelle il travaille et partager l'étape de création avec un collègue. La liberté du trait est apportée par EsQUIsE ou SketSha mais il reste à fournir une navigation dans le document virtuel qui impose le moins de contrainte possible. Le concepteur doit pouvoir manipuler la feuille de papier virtuelle aisément pour la déplacer ou la tourner. Cette feuille ne peut déborder physiquement de la surface du Bureau Virtuel et il est nécessaire également de pouvoir visualiser l'ensemble du support ou seulement une partie. Lorsque le concepteur regarde le modèle 3D formé à partir de ses croquis, il peut désirer l'inspecter de l'intérieur ou au contraire le dominer pour le voir de l'extérieur. Ou encore, il peut vouloir s'y déplacer comme s'il marchait à proximité ou dans le bâtiment.

1.4 Programmation au Lucid Group

Comme mentionné dans la section précédente, la programmation au Lucid Group se fait en Java. Pour la programmation 3D, la bibliothèque Java 3D est utilisée. Une des premières tâches dans le laboratoire de recherche a été d'évaluer la faisabilité technique de prérequis pour les interfaces envisagées (les idées les concernant se sont précisées petit à petit par la suite) et d'apporter la preuve par l'exemple. Les problèmes à surmonter concernaient essentiellement l'affichage en surimpression d'un widget par dessus une scène 3D et les changements de points de vue en fonction des coordonnées du point de la scène désigné par le curseur à l'écran (voir la notion de *picking*).

Le code écrit pour la navigation est utilisé par, et utilise, du code produit par les programmeurs du Lucid Group. En particulier, SketSha a été utilisé pour une expérience de cours de trois mois où des étudiants en architecture belges travaillaient avec des étudiants français (les uns et les autres avaient accès à un Bureau Virtuel et à un système de visioconférence). Une version du widget de navigation a alors été utilisée le dernier mois, ce qui a permis de récupérer de façon informelle quelques avis d'étudiants.

1.5 Contraintes d'interactions

Nous sommes concerné par les différentes possibilités de navigation dans une scène 3D ou sur un plan 2D ainsi que les moyens de les activer. Le contexte d'utilisation contraint ces possibilités. L'interaction avec le logiciel se fait au stylet (sur le Bureau Virtuel). Nous excluons tout autre mode (écran tactile, voix, ...). Nous nous limitons à l'utilisation de la pointe du stylet ; nous n'utilisons pas de bouton se trouvant sur le corps du stylet ou l'extrémité opposée (correspondant à une gomme sur un crayon). En outre, un seul stylet à la fois est utilisé sur la surface (mais il peut être partagé par un groupe de personnes collaborant sur le Bureau)[2].

Une contrainte importante supplémentaire (généralement absente dans les recherches précédentes) est que les opérations de navigation cohabitent avec une opération de dessin dans le cas 2D. Plus précisément, il n'existe pas deux modes, l'un pour le dessin et l'autre pour la navigation, qui permettraient de disposer librement de tous les évènements de pointage (mouvement du pointeur, pression du bouton, ...). Par exemple, si l'on considère que le dessin est l'opération dominante, un trait du stylet doit dessiner un trait sur la feuille virtuelle. De même, est-ce qu'un simple click doit produire un point sur la feuille virtuelle ou bien peut-on utiliser le click pour faire apparaître un menu ? Ou encore peut-on se permettre un menu visible en permanence permettant de passer en mode navigation ?

Dans le cas 3D, il est demandé que le click dans la scène ne soit pas utilisé pour les opérations ou changements de mode de navigation (mais contrairement à l'application de dessin, le trait est disponible). Il s'agit d'une restriction importante puisque le click est l'interaction la plus utilisée (ce qui justifie que cet évènement soit réservé par l'application dans laquelle est déployé le widget de navigation).

[2]Du point de vue du programmeur, un stylet et une souris sont semblables et les évènements logiciels générés par le matériel sont identiques. De ce fait, les termes stylet et souris sont interchangeables, de même que pointe du stylet ou bouton.

Chapitre 2

Navigation 2D et 3D

La navigation dans une application 2D ou 3D comporte deux aspects. Le premier concerne les opérations que l'application offre. Par exemple, est-il possible d'effectuer une rotation ? Et dans ce cas, où se trouve le centre de rotation ? Le second aspect concerne la manière dont les opérations disponibles sont utilisées, c'est-à-dire comment le mode associé à une opération est-il activé ou désactivé et que doit faire l'utilisateur pour employer le mode. Par exemple, dans une application 2D, déplacer la feuille virtuelle se fait avec l'outil de *panning*, souvent symbolisé par une main avec les doigts à plat. Pour utiliser cet outil, faut-il cliquer sur un bouton dans un menu, ou se servir d'un bouton particulier de la souris ? Ce chapitre traite du premier aspect, d'abord dans le cas d'une scène à deux dimensions, puis dans le cas d'une scène à trois dimensions.

Dans une application 2D, la visualisation d'une scène dépend de la position, de la taille et de l'orientation d'un rectangle placé dans la scène. Ce rectangle, nommé (sous-)rectangle de visualisation est tout ce qu'il faut pour déterminer ce qui est affiché à l'écran. Naviguer dans la scène revient alors à modifier la position, la taille ou l'orientation de ce rectangle.

La visualisation d'une scène à trois dimensions ne nécessite pas un rectangle de visualisation mais bien un volume de visualisation placé dans la scène. Ce volume est soit un parallélépipède rectangle, soit une pyramide tronquée (à six faces), appelée *frustum*[FvDFH96]. Comme dans le cas 2D, manipuler la position, la taille et l'orientation de ce volume est tout ce qu'il faut pour naviguer dans la scène.

2.1 Scène 2D

Nous présentons d'abord l'environnement de travail à deux dimensions. C'est dans le cadre d'une scène 2D qu'il a été prévu d'évaluer différentes techniques de changement d'opération de navigation. La scène peut être assimilée à R^2 mais nous considérons le cas où elle est de taille finie et de forme rectangulaire. Visualiser la scène consiste à afficher à l'écran[1] un sous-rectangle de la scène (fig. 2.1). Ce sous-rectangle possède les mêmes proportions que l'écran (ou la fenêtre de l'application)[2]. Le sous-rectangle visualisé peut varier de trois manières : en position, en taille et en orientation.

Cette description correspond naturellement à beaucoup d'utilisations de programmes. Entre autres, il s'agit de la manière dont une application de retouche photo est affichée. Si l'image est plus grande que l'écran, seule une portion de cette image est visible et l'application offre la possibilité de déplacer l'image pour en voir le reste. Dans certains cas, une vignette montre une version réduite de toute l'image ainsi qu'un rectangle qui indique la portion de l'image visible.

Changer la position du sous-rectangle permet de voir une autre portion de la scène. Varier la taille du sous-rectangle donne l'impression à l'écran de (dé)zoomer sur la scène : un sous-rectangle plus petit affichera une plus petite part de la scène sur l'écran (donc plus de détails sont visibles). Enfin, orienter le sous-rectangle fait tourner la scène à l'écran.

2.1.1 Navigation

Dans l'environnement 2D, naviguer dans la scène revient à changer l'état du sous-rectangle de visualisation. Pour chaque opération décrite, on adopte d'abord le point de vue de la manipulation d'un objet placé dans la scène, puis on indique comment cela affecte la vue de la scène dans le cas où l'objet manipulé est le sous-rectangle de visualisation.

En deux dimensions, la manipulation d'un objet possède quatre degrés de liberté : la translation peut se faire sur l'axe x ou l'axe y, la rotation se fait sur l'axe z et le changement d'échelle uniforme fournit un degré de liberté supplémentaire.

---

[1] Dans ce texte, nous ne parlons que d'application formée d'une seule fenêtre et par la suite, nous ne faisons pas la différence entre écran ou fenêtre.

[2] De cette manière, les proportions affichées correspondent aux proportions des éléments visibles de la scène. Pour le genre d'application qui nous occupe, déformer l'affichage n'est pas nécessaire.

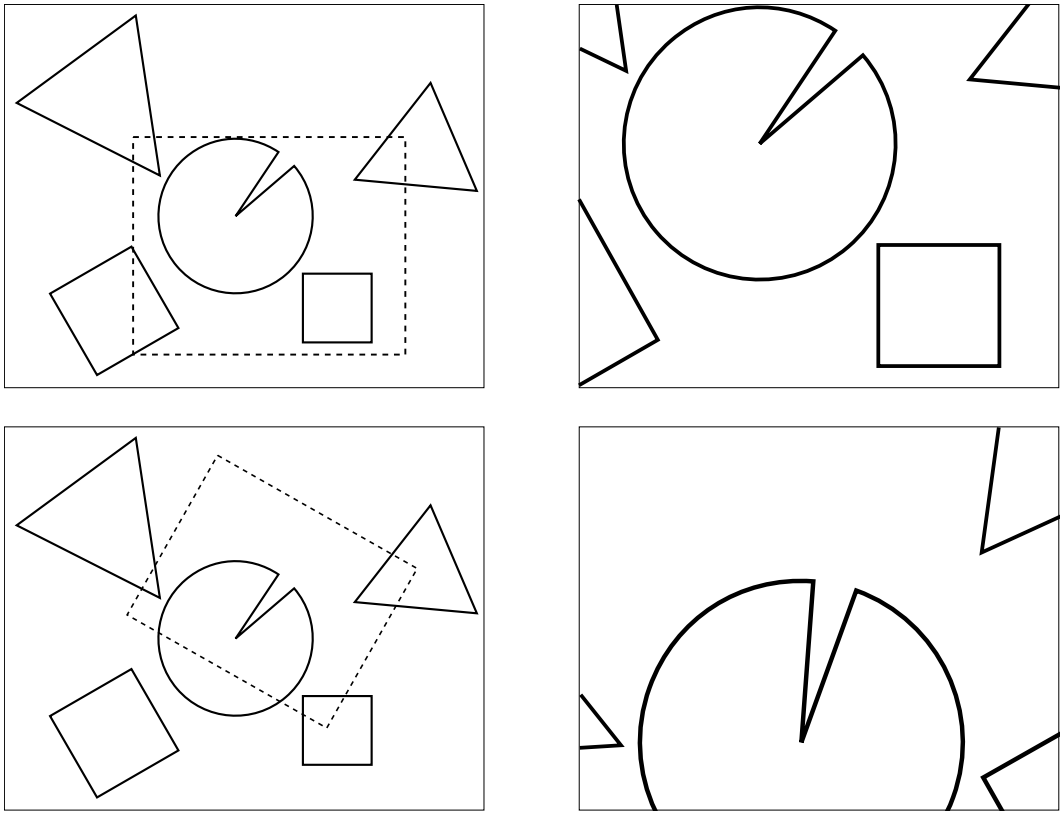


FIG. 2.1 – *La colonne de gauche représente deux fois la même scène avec le rectangle de visualisation placé de manières différentes (en traits pointillés). La colonne de droite représente la partie de la scène visible à l'écran pour chacune des deux configurations. Dans les deux images du bas, par rapport à celles du haut, le rectangle a été déplacé, tourné et réduit de taille. A l'écran, le déplacement et la rotation de la scène sont inversés par rapport à ceux du rectangle et la scène paraît plus grande.*

Translation

La translation d'un objet dans le plan correspond naturellement au déplacement à deux dimensions du dispositif de pointage. Pour rendre l'interaction intuitive, il est préférable que le curseur garde sa position relative à l'objet pendant la translation. Ceci est d'autant plus vrai lorsque le dispositif de pointage est un stylet utilisé sur la surface d'affichage. Pour des surfaces d'affichage de très grande taille, il est peut être justifié d'utiliser une ampleur de mouvement relative de façon à déplacer un objet graphique sur une grande distance sans que l'utilisateur n'ait à se déplacer. Mais pour une surface de la taille d'une table comme dans ce travail, la première solution est préférée. Autrement dit, un déplacement dt du curseur à l'écran donne un déplacement dt à l'écran de l'objet manipulé (exactement comme avec une feuille de papier sur un bureau que l'on déplacerait du bout du doigt) (fig. 2.2).

La caméra (le rectangle de visualisation) peut être manipulée de façon similaire à un objet quelconque. Cependant, alors que l'objet peut être manipulé directement à l'écran, la caméra n'est pas visible. La manipulation directe de la caméra peut se faire en manipulant l'entièreté de la scène : déplacer la scène vers la gauche ou déplacer la caméra vers la droite aboutit dans le même changement de vue à l'écran. Puisqu'il n'y a pas d'autre point de repère, les deux interprétations sont interchangeables. Néanmoins, comme le déplacement d'une feuille de papier devant nous nous est familier (et qu'il est plus rare de nous déplacer parallèlement à une surface en gardant le regard perpendiculaire à cette surface), l'interprétation la plus naturelle est le déplacement de la scène.

Rotation

La rotation d'un objet en deux dimensions peut se faire uniquement sur l'axe des z, l'axe perpendiculaire à la scène, ou, si l'on préfère, l'axe qui "sort" de l'écran. Il est pratique pour l'utilisateur de pouvoir décider du centre de rotation mais cela n'est pas techniquement nécessaire : toute configuration position/rotation d'un objet dans le plan peut être obtenue par composition de translations et de rotations. Par exemple, si les rotations ont toujours lieu avec le centre de la scène comme centre de rotation, faire tourner un objet sur lui-même peut être obtenu en translatant l'objet au centre de la scène, faire tourner l'objet (dont le centre coïncide avec le centre de la scène et donc aussi avec le centre de rotation), puis le translater à nouveau à sa position initiale (fig. 2.3).

Même s'il est possible d'obtenir le résultat voulu, pour simplifier l'opération de rotation, permettre de décider où se trouve le centre de rotation est nécessaire. Le centre de rotation est conceptuellement un objet qui peut seulement être translaté.

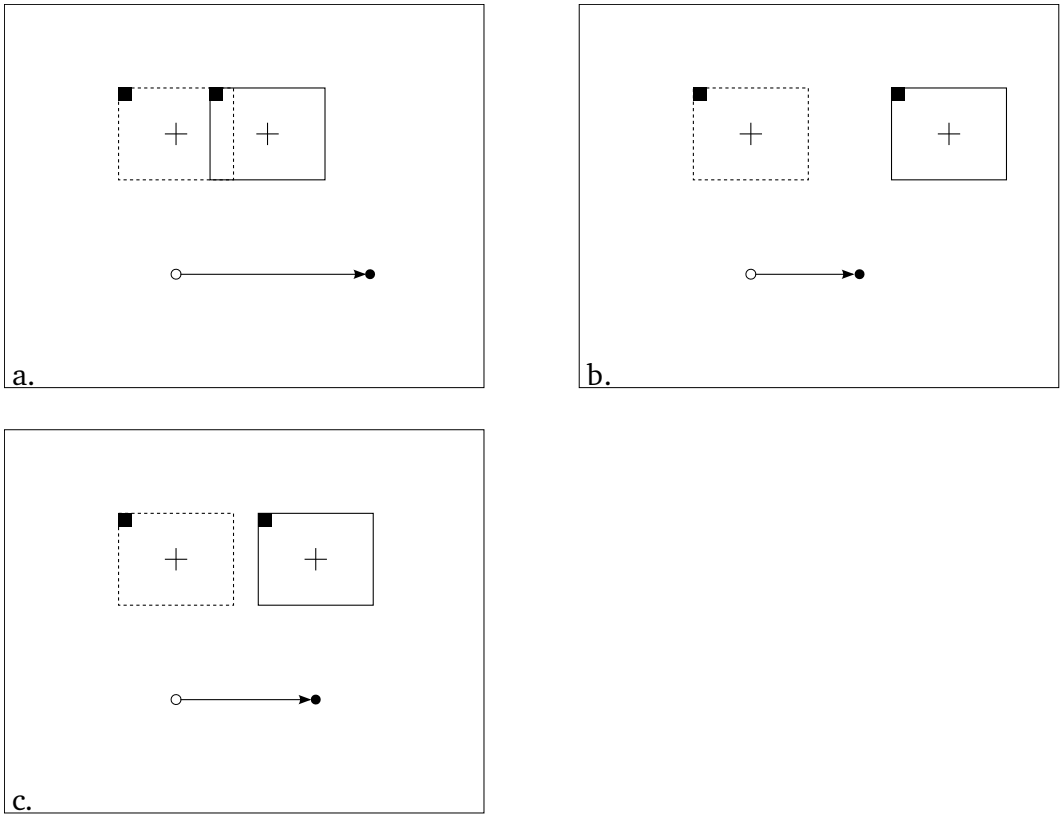


FIG. 2.2 – *Différents rapports entre le mouvement du curseur (flèche vers la droite) et celui de la scène, qui contient ici seulement un rectangle. Dans (a), le mouvement de la scène suit le curseur mais avec une ampleur moindre et dans (b), il le suit avec une ampleur plus importante. Dans (c), le mouvement de la scène correspond exactement à celui du curseur.*

Comme pour la translation, la rotation de la caméra peut être interprétée comme une rotation de la scène et pouvoir décider du centre de rotation est également souhaitable.

Mise à l'échelle

L'opération de mise à l'échelle d'un objet est semblable à l'opération de rotation. Un seul degré de liberté est nécessaire (parce que nous nous limitons à une mise à l'échelle uniforme plutôt que de permettre de changer l'échelle des axes x et y indépendamment) et la notion de centre existe également. Le centre de mise à l'échelle est le point fixe de la transformation (fig. 2.4). Comme pour la rotation, il

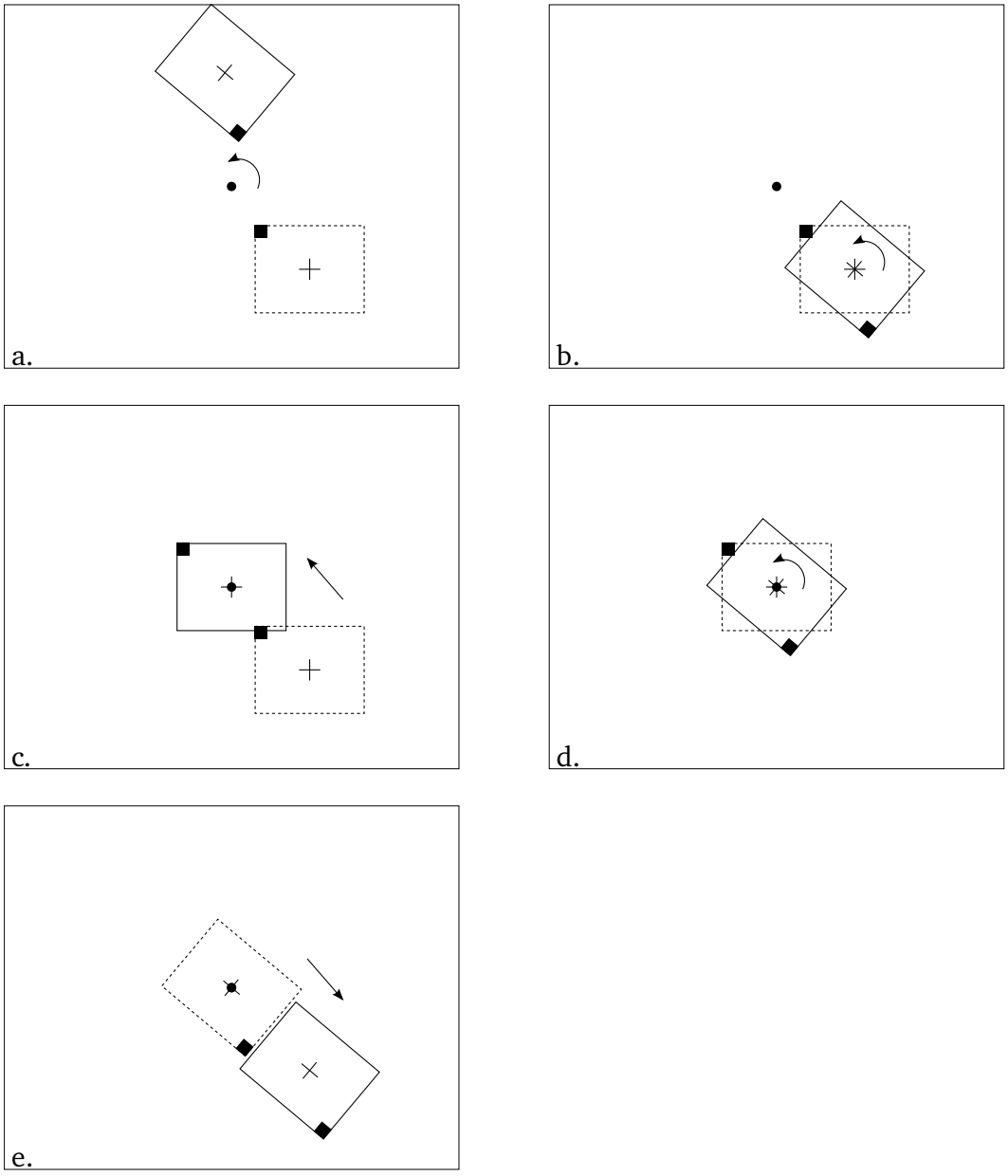


FIG. 2.3 – *La position avant l'opération est marquée en pointillé et la position après en trait plein. Le centre de la scène est symbolisé par le point et le centre du rectangle par une croix. Le carré noir permet de visualiser la rotation. (a) La rotation du rectangle autour du centre de la scène de 140° change également sa position. (b) Par contre, la rotation autour du centre du rectangle ne le déplace pas. Il est possible d'obtenir la rotation de (b) en combinant translations et rotation autour du centre de la scène. (c) D'abord, une translation amène le centre du rectangle au centre de la scène. (d) Ensuite, une rotation est effectuée. Pour finir, le rectangle est ramené à sa position initiale par une seconde translation (e). La position initiale et la position finale dans (b) correspondent à la position initiale dans (c) et à la position finale dans (e).*

n'est pas nécessaire techniquement de pouvoir décider de sa position mais cela simplifie considérablement les manipulations que l'utilisateur doit réaliser pour obtenir le résultat voulu.

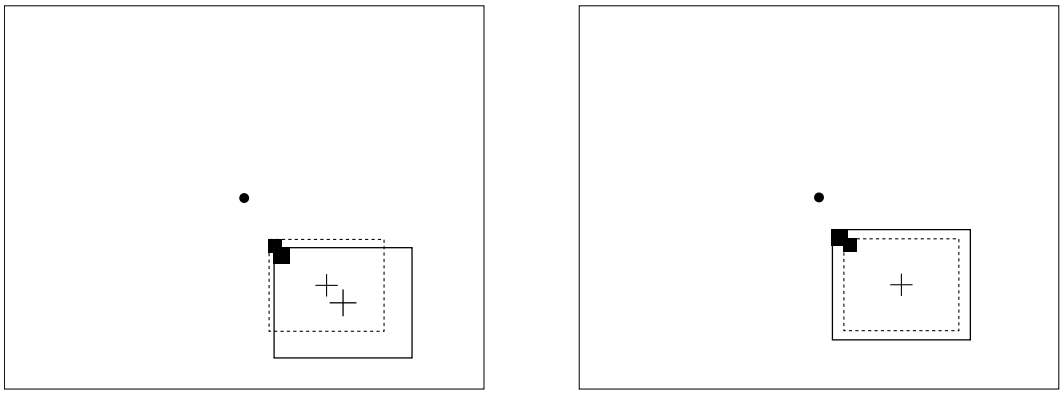


FIG. 2.4 – *La mise à l'échelle possède comme la rotation une notion de centre. A gauche, la mise à l'échelle se fait par rapport au centre de la scène et le centre du rectangle est déplacé. A droite, la mise à l'échelle est effectuée par rapport au centre du rectangle et celui-ci reste sur place.*

Il est possible de modifier l'échelle du rendu de la scène à l'écran en changeant la taille du rectangle de visualisation. A nouveau, en deux dimensions, il est préférable de pouvoir choisir le centre de la mise à l'échelle.

2.2 Scène 3D

L'affichage d'une scène 3D sur un écran dépend de la position du point de vue adopté sur la scène. Il est courant et commode de parler de la position d'une caméra virtuelle[3] dans la scène à visualiser pour déterminer le résultat à l'écran. Comme dans le cas 2D, l'emplacement de la caméra dans la scène affecte la vue obtenue.

2.2.1 Projection perspective et projection orthographique

Dans la section précédente, la vue sur la scène 2D dépend d'un rectangle placé dans cette scène. En trois dimensions, l'affichage dépend d'un volume de visualisation à six faces : soit un parallélépipède rectangle, soit un frustum (pyramide dont le sommet est tronqué). La forme du volume de visualisation influence le rendu de la

[3]Pour simplifier, nous parlerons par la suite de caméra sans mentionner qu'elle est virtuelle.

scène. Il s'agit d'un parallélépipède rectangle dans le cas d'une projection orthographique et d'un frustum dans le cas d'une projection perspective (fig. 2.5). La position du volume est donnée par la position de la caméra. La partie de la scène contenue dans le volume est projetée en direction du point de vue, c'est-à-dire la position de la caméra (le sommet de la pyramide dans le cas de la projection perspective), sur le plan proche.

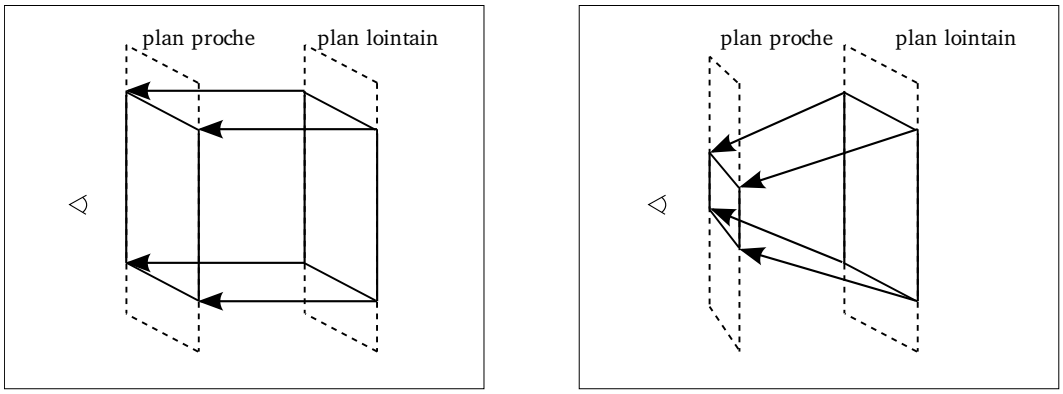


FIG. 2.5 – *Projection orthographique et projection perspective. L'image est formée sur la face du volume de visualisation située sur le plan proche. La taille d'un objet dans la scène n'est pas influencée par la distance entre l'objet et le point de vue dans la projection orthographique alors qu'elle l'est dans la projection perspective.*

Pour une projection orthographique, la distance d'un objet par rapport au point de vue n'influence pas la taille qu'il prend à l'écran (la taille du volume ne varie pas en fonction de son éloignement à la caméra) alors que dans le cas de la projection perspective, un objet couvre une portion de l'écran moins importante s'il est plus éloigné (la taille du volume de projection croît au fur et à mesure que l'on s'éloigne du point de vue). La projection perspective est utile pour produire une image d'une scène de façon réaliste car elle imite le fonctionnement d'une véritable caméra ou de nos yeux. La projection orthographique est néanmoins aussi utile lorsque l'on souhaite pouvoir comparer à l'écran les dimensions des objets de la scène.

2.2.2 Navigation

Un objet virtuel placé dans une scène 3D possède six degrés de liberté : la translation qui peut se faire sur un des trois axes (x, y et z) et la rotation qui peut être définie en fonction d'angles sur ces trois axes. Il en est bien entendu de même pour la caméra que nous utilisons pour visualiser la scène. Les opérations de translation et de rotation sont similaires au cas 2D et ne sont pas détaillées à nouveau. L'opération

de zoom par contre mérite une explication supplémentaire. En principe, zoomer ne change pas le point de vue. Ce qui est devant la caméra avant le zoom, reste devant la caméra après. Cependant, le mot zoom est souvent employé pour désigner une translation sur l'axe des z (l'axe perpendiculaire à l'écran).

Pour les mouvements de caméra dans une scène 2D, nous avons indiqué que l'interprétation du mouvement à l'écran correspondait plutôt au déplacement de la scène (comme une grande feuille de papier) plutôt qu'à celui d'une caméra. En trois dimensions, il y a toujours deux interprétations possibles. Celle qui est choisie dépendra fortement du contenu de la scène et de la manière dont l'utilisateur doit contrôler la navigation. Etant donné le contexte de ce travail, nous sommes concerné par la visualisation de bâtiments. Il peut s'agir d'un seul bâtiment ou d'une portion de ville : plusieurs bâtiments avec des routes, des ponts, ... Si l'utilisateur domine la scène (elle apparaît en entier à l'écran), il aura davantage l'impression de la manipuler. Par contre, s'il est immergé dans la scène, il aura l'impression de se déplacer à l'intérieur plutôt que de la manipuler. Cette impression peut être renforcée ou diminuée suivant l'interaction nécessaire pour déplacer la caméra. Prenons un exemple : la scène consiste en un cube affiché à l'écran et le déplacement du point de vue s'effectue avec les touches fléchées du clavier. Si presser la flèche gauche déplace la caméra à droite, cela donne l'impression que la flèche contrôle le cube (puisqu'à l'écran, il se déplace vers la gauche). Si cela déplaçait la caméra vers la gauche, cela donnerait l'impression que c'est la caméra qui est contrôlée.

2.2.3 Métaphores de navigation

L'exemple de la section précédente illustre en fait les métaphores *scene-in-hand* et *eyeball-in-hand* [WO90]. Dans la première métaphore, l'utilisateur contrôle la scène et considère que le point de vue est fixe. Dans la seconde métaphore, l'utilisateur contrôle l'oeil par lequel il voit la scène et celle-ci est considérée comme fixe. Cette distinction est valable autant pour la translation que pour la rotation.

Dans une scène comportant des bâtiments, on peut envisager de regarder les bâtiments uniquement de l'extérieur (comme si l'on se promenait dans les rues autour de ces bâtiments) ou également de l'intérieur. On peut souhaiter que l'interaction se fasse comme si l'on marchait sur le sol ou plutôt être libre d'adopter tout point de vue (comme un oiseau). Le cas où l'on simule un déplacement à pied est bien connu puisqu'il correspond aux jeux vidéos d'action à la première personne. Ce terme signifie que l'image rendue est similaire à celle que le personnage incarné verrait. Par comparaison, la vue à la troisième personne présente le personnage contrôlé visible à l'écran, généralement de derrière et légèrement du dessus.

Quel que soit le type de changement de vue utilisé, le mouvement peut être ou non

limité de différentes façons. La contrainte la plus courante est l'impossibilité dans les jeux vidéos de passer à travers les murs. Une autre contrainte, importante dans le cas de l'architecture, est liée à la notion de verticalité : en voyant une image de bâtiments à l'écran, il est possible d'indiquer immédiatement si l'image est à l'envers ou si elle penche légèrement. Il n'y a que si l'on voit la scène à la verticale que la question ne se pose pas. Pour cette raison, faire une rotation de la caméra doit généralement conserver les verticales.

Chapitre 3

Etat de l'art et logiciels existants

Ce chapitre comporte deux sections principales. La première traite de la façon dont trois logiciels populaires de modélisation 3D permettent la navigation dans une scène virtuelle. La seconde section décrit les principales réalisations en matière de techniques d'interactions susceptibles d'être utilisées au stylet sur une surface d'affichage assez grande (le Bureau Virtuel). C'est sur base de ces techniques que le choix des widgets de navigation à implémenter a été fait avec les membres du Lucid Group.

3.1 Navigation dans les logiciels 3D existants

La navigation dans une scène est semblable d'un logiciel à l'autre, tant en terme de mouvements possibles qu'en terme de leur mise en oeuvre par l'utilisateur. Tous les logiciels fonctionnent selon le principe de mode. Différents modes sont disponibles. Parmi ces modes, on trouve les outils de sélection, de manipulation, de changement d'apparence ou de géométrie. La navigation dans la scène virtuelle est également réalisée via l'utilisation de modes spécifiques. Un mode est souvent symbolisé par l'utilisation d'un curseur spécifique. Il peut être sélectionné par une entrée dans un menu ou par l'utilisation de touches du clavier ou de boutons de la souris (éventuellement par une combinaison de touches ou de boutons). Dans le cas de la sélection via un menu, le mode choisi est permanent, c'est-à-dire qu'il ne sera remplacé que par la sélection d'un autre mode. Au lieu d'être permanent, un mode peut être transitoire (en anglais, *transient*) : il dure seulement tant que le bouton ou la touche d'activation du mode est maintenu enfoncé.

Les modes concernant la navigation sont fort semblables et sont détaillés dans les

sections qui suivent. Leur mise en oeuvre varie très légèrement sur le choix des touches ou boutons employés mais le principe reste le même.

Les logiciels examinés sont 3D Studio Max, Maya et SketchUp. Ces logiciels sont destinés à être utilisés avec un clavier et une souris à trois boutons minimum. Le problème de limitation des moyens de passage d'un mode à un autre est donc beaucoup moins important que dans le contexte du Bureau Virtuel et du stylet (où l'utilisateur ne manipule que l'équivalent d'une souris à un seul bouton, sans clavier).

3.1.1 Opérations standards

Chacun des logiciels examinés propose trois opérations de navigation que l'on peut considérer comme le standard de la navigation. Ces trois opérations sont le *panning*, le zoom et la rotation.

Panning

Le panning, que l'on retrouve dans les logiciels de traitement 2D, permet de déplacer la caméra parallèlement au plan de projection. Puisque le mouvement se fait dans un plan, il peut correspondre au mouvement de la souris ou du stylet : un mouvement vertical (resp. horizontal) du curseur réalise un mouvement vertical (resp. horizontal) de la caméra. Dans une application 2D, il donne l'impression de déplacer le support de travail.

Zoom

Le zoom, également disponible en 2D, permet d'avoir une image agrandie d'une partie de la scène. Bien que dénommé zoom, dans une scène à trois dimensions vue par une caméra perspective, il s'agit plutôt d'un mouvement de la caméra suivant un axe perpendiculaire à l'écran. Ce mouvement est parfois appelé *dolly* en référence au mouvement similaire effectué au cinéma. Si la caméra donne lieu à une projection orthographique, déplacer la caméra sur l'axe perpendiculaire à l'écran ne change pas l'image formée (des objets de la scène peuvent se retrouver derrière la caméra et ne plus être visibles mais peu importe la distance d'un objet, sa taille à l'écran restera la même). Dans ce cas, le terme de zoom est plus approprié.

Rotation

La troisième opération consiste à faire tourner la caméra autour d'un point spécifique de la scène défini de façon différente d'une application à l'autre. Pour préciser que la caméra tourne autour d'un point plutôt que sur elle-même, cette opération est souvent appelée en anglais *arc-rotate* ou *orbit*. Généralement, la position du centre de rotation dépend de la position de la caméra, de la partie de la scène visible ou de la partie de la scène sélectionnée.

Opérations supplémentaires

Les trois opérations décrites ci-dessus sont celles les plus employées. Pour obtenir le point de vue désiré sur la scène, l'utilisateur est généralement amené à les utiliser à tour de rôle, passant rapidement d'un mode à un autre. Ces opérations sont complétées par d'autres moins souvent utilisées. Il est souvent possible de centrer la vue sur un objet sélectionné de manière à ce qu'il remplisse presque toute la surface d'affichage. Ou encore, une opération *walkthrough* permet de se déplacer sur un plan horizontal de façon continue, l'orientation de la souris permettant de s'orienter en tournant selon un axe vertical.

3.1.2 Point de référence et picking

Les opérations de navigation ne sont pas définies uniquement en fonction de la position et de l'orientation de la caméra mais peuvent nécessiter un point de référence. Elles peuvent dépendre de ce qui est sélectionné, de ce qui est visible ou de l'endroit où se trouve le curseur. La position du curseur peut correspondre à l'emplacement d'un objet à l'écran. Le point précis de l'objet ainsi désigné possède des coordonnées dans l'espace de la scène. Il est donc possible, à partir des coordonnées à l'écran (en deux dimensions) du curseur d'obtenir les coordonnées 3D d'un point dans la scène. Ce travail est réalisé par une fonctionnalité de la bibliothèque 3D appelée *picking*.

Le picking est réalisé en calculant un rayon partant du point de vue et dont la direction dépend de la position du curseur. Ce rayon est ainsi dirigé vers le point de la scène qui est projeté au même endroit que le curseur (voir la description du frustum et la figure 3.1). L'intersection de ce rayon et de la scène permet de connaître l'objet pointé par le curseur ainsi que les coordonnées d'intersection. Le point d'intersection peut alors servir de point de référence pour certaines opérations.

Un point de référence est nécessaire pour les opérations du type arc-rotate ou orbit. Ces opérations font tourner la caméra autour d'un point. Par exemple, il peut s'agir

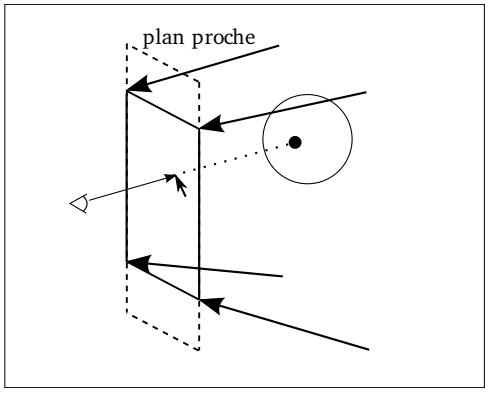


FIG. 3.1 – *Cette figure illustre le principe du picking. A partir du point de vue (tout à gauche), il est possible de tracer un rayon qui part en direction du volume de visualisation (vers la droite) et qui passe par la position du curseur sur l'image formée sur le plan proche. Le calcul de l'intersection la plus proche du point de vue entre ce rayon et la scène donne un point de référence pour certaines opérations de navigation.*

d'un objet sélectionné. Le zoom peut aussi être effectué par rapport à un point de référence. Ce point est alors fixe à l'écran pendant l'opération de zoom (voir la description du zoom en deux dimensions).

Le panning peut également bénéficier d'un point de référence. Le panning peut par exemple être réalisé par rapport à un objet et maintenir le point d'intersection du rayon défini par le curseur et cet objet fixe pour obtenir un effet de manipulation directe semblable au déplacement d'une feuille de papier du bout du doigt (le doigt et la feuille restent fixes l'un par rapport à l'autre). Si l'objet n'est pas déterminé (il n'y en a pas sous le curseur), il peut être conceptualisé comme un plan parallèle à la vue et à une certaine distance. La distance de ce plan (nommé plan de panning dans la suite) peut être la distance du dernier objet manipulé ou le centre de la portion de scène qui est visible.

SketchUp

Des quatre applications, SketchUp est la seule dont le centre de rotation et le plan de panning sont dépendants de la partie de la scène qui est visible à l'écran. Ainsi, si un seul objet est visible, l'arc-rotate permet de tourner autour de cet objet sans le sélectionner explicitement au préalable.

Dans le cas où plusieurs objets sont visibles, leur importance dans la vue (s'ils sont plus ou moins éloignés à l'écran) est prise en compte et la rotation se fera autour de

plusieurs objets.

Une situation moins intuitive se présente lorsque, avant l'utilisation de l'arc-rotate, un seul objet est visible à l'avant plan et qu'un second apparaît à la vue pendant la rotation. Au début, la rotation garde le premier objet bien centré puis le centre de rotation se déplace lorsque le deuxième objet devient visible (à moins qu'il ne soit fort éloigné) et donc le premier objet n'est plus au centre de l'écran.

Le panning utilise le principe de manipulation directe et le plan de panning est situé à la profondeur du point donné par le picking s'il existe ou par la position des objets proches s'il n'existe pas.

3D Studio Max et Maya

Le centre de rotation peut être, en fonction de la variante de l'arc-rotate choisie, la sélection (objet, un groupe d'objets ou un sous-élément d'un objet) ou être lié à la vue. Dans ce cas, il est situé sur l'axe des z, côté négatif. Lorsque l'application démarre, la caméra est dirigée vers le centre de la scène. L'opération de rotation et celle de zoom déplacent la caméra par rapport à ce centre.

3.1.3 Panning

SketchUp

Dans SketchUp, le plan de panning est situé à l'intersection donnée par le picking lorsqu'elle existe. Le principe de manipulation directe possède un aspect particulièrement intéressant dans le cas du panning dans une vue en perspective. La perspective raccourcit les distances. Un objet de taille constante apparaît à l'écran de plus en plus petit lorsqu'il s'éloigne du point de vue. Inversement, le panning avec manipulation directe peut effectuer un plus grand déplacement de la caméra si le plan de panning est situé plus loin. L'utilisateur peut donc varier intuitivement l'ampleur du mouvement de caméra pour un même mouvement du curseur suivant la distance de l'objet intersecté par le picking. Tout se passe comme si l'utilisateur piquait l'objet avec une punaise et le plaçait où il voulait à l'écran (fig. 3.2).

Lorsque l'outil zoom est sélectionné, un double-click fait un panning instantané qui a pour effet de centrer à l'écran le point de picking.

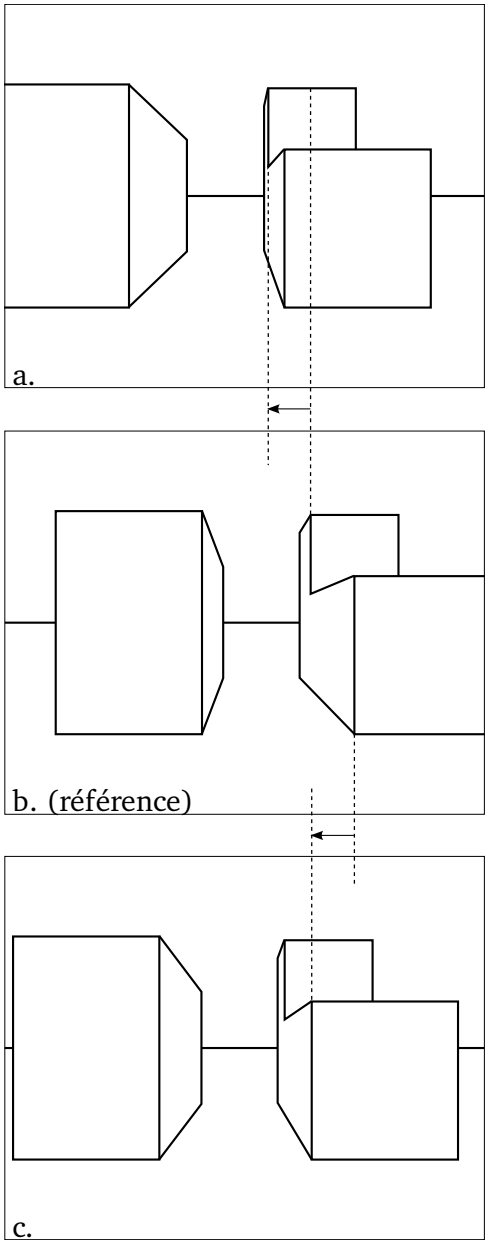


FIG. 3.2 – *Cette figure illustre les mouvements de caméra obtenus par un panning avec manipulation directe en 3D. La situation de départ est visible en (b). (a) présente la situation après un panning effectué avec le curseur sur la face partiellement visible du volume de droite. (b) présente la situation après un panning effectué avec le curseur sur la face à l'avant-plan. Les flèches indiquent le mouvement du curseur à l'écran. La distance parcourue par le curseur est identique pour les deux pannings mais le mouvement de caméra est plus important en (a) qu'en (b) à cause du raccourcissement dû à la perspective.*

3D Studio Max et Maya

Dans 3D Studio Max et Maya, la profondeur du plan de panning correspond à celle du centre lié à la vue (il s'agit du même point que celui utilisé pour la rotation).

3.1.4 Zoom

SketchUp

SketchUp offre deux possibilités pour réaliser un zoom (une translation sur l'axe des z). L'un est activé par une entrée dans un menu (mode permanent) et l'autre par la molette de la souris. Le zoom utilisé par le menu fait avancer ou reculer la caméra sur l'axe des z. Le zoom avec la molette déplace la caméra le long du rayon de picking. Ce déplacement conserve le point de picking sous le curseur. Il est donc pratique lorsque l'on désire zoomer sur un endroit qui n'est pas situé au centre de l'écran. Autrement, il faudrait alterner zoom et panning pour obtenir le même effet.

3D Studio Max

Le facteur de zoom est proportionnel à la distance entre le point de vue et le point de référence au moment du début du zoom. Cela signifie que lorsque la caméra est située proche du centre, un mouvement de souris plus important sera nécessaire pour un même déplacement de la caméra. Aussi, en démarrant de plus loin, avec un seul mouvement de souris, il est possible de déplacer la caméra plus loin que le centre (parce que le mouvement de la caméra est fort important). Si le zoom est fait en plusieurs petits mouvements, c'est le cas par exemple en utilisant la molette, le mouvement finit par devenir nul (au fur et à mesure que l'on se rapproche du centre, le mouvement de la caméra diminue). Il faut alors de nombreuses utilisations de cet outil pour sortir de cet état (le recul aussi est affecté par cette proportionnalité).

Maya

Dans Maya, le facteur de déplacement ne dépend pas de la distance du centre de rotation. Contrairement à 3D Studio Max, si la distance au centre devient presque nulle, l'amplitude du mouvement n'en sera pas réduit pour la cause.

3.1.5 Arc-rotate

SketchUp

Il existe différentes manières d'implémenter l'idée de la rotation de la caméra autour d'un point. Dans [Hul90], un mouvement de souris qui fait une boucle (le curseur revient à son point de départ) ne remet pas la caméra dans son état initial et son utilisation n'est pas intuitive. [Sho94] par contre est indépendant du chemin parcouru par le curseur et seuls les points de départ et d'arrivée du chemin importent. Malheureusement, il est difficile de conserver la verticale de la scène à l'écran[LGC99]. Dans SketchUp, la verticale est préservée : peu importe le mouvement réalisé par la souris, la verticale de la scène apparaîtra verticale à l'écran (sauf dans le cas extrême où la caméra pointe verticalement). Il s'agit d'une propriété intéressante dans ce genre d'application. En effet, SketchUp est principalement utilisé pour modéliser des bâtiments. La notion de verticalité est donc importante. Par contre, rien n'empêche de faire une rotation qui place la caméra au-dessus de la scène, en regardant vers le bas, et de dépasser ce point. La caméra est alors placée à l'envers.

3D Studio Max

L'outil de rotation dans 3D Studio Max permet de choisir plusieurs effets. Lorsque le mode de rotation est utilisé, un cercle avec quatre poignées est visible au centre de l'écran. Un déplacement dans le centre de ce cercle réalise un arc-rotate. En dehors du cercle, la caméra tourne le long de l'axe z (comme si l'on penchait la tête sur le côté). Les poignées permettent de faire un arc-rotate contraint sur l'axe vertical ou sur l'axe horizontal.

3.2 Etat de l'art

Cette section présente un état de l'art en matière de techniques d'interactions adaptées à l'utilisation d'une grande surface d'affichage. Elles concernent toutes l'utilisation d'un dispositif de pointage simple comme la souris ou la tablette graphique.

Les techniques sont relativement simples. Il existe d'autres possibilités plus complexes à mettre en oeuvre, telles que la reconnaissance de gestes, de traits[1] ou de

[1]Par gestes, nous entendons des mouvements faits avec le dispositif de pointage. Ils peuvent être effectués avec le bouton appuyé (ou avec contact entre le stylet et la surface tactile) ou non. Du point de vue du moteur de reconnaissance, seul le chemin du curseur importe et la reconnaissance de gestes ou de traits est donc similaire.

voix. Dans [FKP+03], les auteurs font mention de ces techniques mais les écartent en faveur de méthodes d'interactions plus traditionnelles, parce que les calculs nécessaires pour traiter la reconnaissance de gestes ou de la voix gêneraient le passage rapide d'un outil à un autre. Nous trouvons l'argument léger et justifions l'exploration de techniques basées sur un simple dispositif de pointage pour des raisons différentes. Ainsi, parmi les techniques existantes, il n'est pas possible de connaître *a priori* la meilleure technique et il est préférable d'avoir à sa disposition plusieurs techniques selon les situations. Ensuite, le fait qu'il soit plus facile d'implémenter ces techniques plutôt que des techniques à base de reconnaissance de traits ou de gestes permet d'en expérimenter davantage, voire de viser une certaine exhaustivité. Enfin, le contexte matériel du Bureau Virtuel nous impose un système de pointage simple. Il n'est donc pas question ici d'interactions à deux mains (voir par exemple [BSF+95]).

3.2.1 Les marking menus

Les *marking menus* [Kur93] [KB94] permettent à un utilisateur de sélectionner une entrée dans un menu. Ils sont utilisables à la souris ou au stylo ; ce dernier donne de meilleurs résultats. Cette sélection peut être réalisée de deux façons distinctes.

La première façon correspond pour une bonne part au fonctionnement d'un menu déroulant classique. Lorsque le stylo est pressé contre la surface d'affichage et gardé dans cet état pendant un court laps de temps (par exemple 200 millisecondes), un menu radial s'affiche. Un menu radial organise ses entrées en cercle. Généralement, les éléments sont équidistants et il est préférable d'en avoir 4, 8 ou 12 placés selon les points cardinaux d'une boussole. La sélection d'une entrée se fait en déplaçant le stylo dessus, tout en le maintenant en contact avec la surface, puis en le levant. Tant que le stylo n'est pas levé, un retour visuel peut indiquer quel élément sera sélectionné lorsque le stylo sera relevé.

La seconde façon d'utiliser un marking menu est de toucher la surface avec le stylo et de réaliser un court trait (ou marque, d'où le nom de cette technique d'interaction) en direction de l'élément que l'utilisateur souhaite sélectionner. Un retour peut indiquer quel élément a été effectivement sélectionné mais il est à noter qu'il n'a pas été nécessaire pour l'utilisateur d'attendre que le menu s'affiche.

C'est l'existence de ces deux fonctionnements qui rend le marking menu si intéressant. Le premier peut être utilisé par l'utilisateur novice et le second par l'utilisateur expérimenté. Le marking menu permet au débutant, d'une part, de découvrir ce que le menu contient et, d'autre part, d'être habitué à réaliser la sélection en faisant le même mouvement physique que l'utilisateur expert. L'utilisateur qui connaît le menu peut rapidement exécuter une sélection sans devoir faire attention mo-

mentanément à l'apparition d'un menu. Le marking menu permet ainsi d'être utilisé aussi bien par un débutant que par un expert et amène le débutant à devenir expert.

Extensions

Plusieurs extensions aux marking menus ont été développées. La plus simple est l'arrangement hiérarchique de menus [KB93]. Ceux-ci sont toujours des marking menus et peuvent être employés comme tels : soit l'utilisateur attend avec le stylo toujours appliqué sur une entrée et le sous-menu radial apparaît, soit il réalise une marque où les changements de direction correspondent aux sous-menus parcourus (la marque est en zigzag).

Dans [GW00], le principe du marking menu est modifié pour permettre la sélection continue de plusieurs éléments (et non pas une seule entrée du menu). Le résultat, baptisé *Flow Menu*, peut par exemple être utilisé pour entrer un nombre en sélectionnant les chiffres qui le composent. Comme pour le marking menu, le stylo démarre du centre d'un menu radial et décrit un trait. Les entrées sont disposées (par exemple) sur les côtés d'un octogone. La sélection n'est pas le côté correspondant à la direction du trait quittant le centre (zone dite de "repos") mais le côté par lequel le trait revient dans la zone de repos. Lorsque le stylo est à nouveau au centre, l'utilisateur peut poursuivre le trait pour sélectionner d'autres côtés. Le nombre d'entrées possibles est augmenté en permettant de placer trois d'entre elles par côté de l'octogone. Celle des trois qui est sélectionnée est déterminée par le trait quittant ce côté soit vers le centre du menu, soit vers un autre côté de l'octogone.

Dans un marking menu hiérarchique, il est possible de remplacer le trait unique (avec changement de direction) par plusieurs traits (sans changement de direction) faits dans un court intervalle de temps [ZB04]. Le but est de permettre de réaliser les traits dans une même zone, indépendamment du nombre de sous-niveaux du menu. Cela permet aussi d'éviter toute ambiguïté lorsque les sous-niveaux parcourus sont dans une seule direction. Une limitation commune au Flow Menu et au marking menu hiérarchique est le nombre d'entrées par niveau qui ne peut excéder raisonnablement huit. En tenant compte de la position de la marque par rapport au centre du marking menu (le point où se situait le stylo au moment où le menu est invoqué), le nombre d'entrées est considérablement augmenté [ZAH06]. La position de la marque est effectivement un premier choix de sous-menu et la direction de la marque un second.

3.2.2 Les interfaces à base de croisements

L'utilisation la plus courante d'un système de pointage comme la souris ou la tablette graphique est de pointer un bouton et de "cliquer". [AZ02] établit les bases pour la réalisation d'interfaces graphiques où l'interaction principale se fait par le croisement du curseur (ou plutôt du chemin établit par le déplacement du curseur sans pression sur un bouton) et une ligne tracée à l'écran. Cette technique est habituellement utilisée pour détecter l'entrée ou la sortie du curseur dans une fenêtre à l'écran. L'idée est appliquée à un logiciel de dessin dans [AG05].

Le principe du croisement est le premier mentionné qui fasse usage du mouvement du curseur sans nécessiter que le stylet soit appuyé sur la surface. Ce principe est intéressant parce qu'il exploite des évènements (les déplacements du curseur) habituellement négligés. Il est à noter que tous les périphériques d'entrée ne permettent pas cette technique : certains écrans tactiles par exemple ne sont pas capables de fournir les coordonnées du curseur sans qu'il y ait un contact avec le stylet.

3.2.3 Tracking Menu et Trailing Widget

Les techniques mentionnées dans les sections précédentes concernent l'interaction entre l'utilisateur et une partie de l'interface graphique. Le sujet de cette section est le *tracking menu* [FKP+03] et le *trailing widget* [FVB06]. La conception de ces deux widgets vise à garder un menu "à portée de click" .

Le tracking menu est un widget correspondant à un menu traditionnel dans lequel le curseur peut se déplacer pour sélectionner une entrée mais qui bougera avec le curseur dès que celui-ci touchera le bord du widget (en direction de l'extérieur).

Plus précisément, le déplacement du curseur qui induit le déplacement du menu se fait sans pression du stylo sur la surface. Autrement dit, cette technique fait usage, comme les interfaces à base de croisements, de la possibilité de suivre la position du stylo sans qu'il y ait contact.

L'utilisation d'un outil se fait en pressant sur l'entrée du menu qui y correspond et en bougeant le curseur (toujours pressé).

L'intérêt de ce genre de menu est bien entendu qu'il reste proche du curseur pour minimiser les aller-retour vers un menu fixe, localisé sur le bord de l'écran. Il est d'autant plus intéressant que le passage d'un outil à un autre est effectué régulièrement ou que la surface d'affichage (et donc la distance à parcourir entre le curseur et le menu) est grande.

Comme le menu est présent en permanence, il est à noter qu'il se prête bien à une utilisation où seule la pointe du stylo permet de cliquer. Il n'est pas nécessaire d'employer le bouton sur le corps du stylo ou un raccourci clavier pour le faire apparaître.

Le tracking menu répond également au besoin d'avoir accès à des fonctionnalités proches du curseur sans avoir recours à un click-droit ou à un raccourci clavier. Ce menu a été développé pour une application utilisée avec un stylo sur un affichage de plusieurs mètres de long. Il s'agit d'un simple bouton rond qui reste à une certaine distance du curseur en temps normal et qui peut être "cliqué" si le mouvement pour l'approcher est suffisamment rapide.

3.2.4 Le hover widget

Le *hover widget* [GHB+06], comme dans la section précédente, a pour but de fournir un menu proche du curseur. Le menu n'est pas visible en permanence et doit être appelé (affiché) par l'utilisateur. C'est dans la manière dont l'appel du menu s'effectue que se trouve l'originalité du hover widget. L'utilisateur réalise un geste (sans contact de la pointe du stylo sur la surface d'affichage), par exemple en forme de 'L' majuscule. A mesure que le geste s'exécute, le widget apparaît en fondu progressif. Si le geste n'est pas terminé (il s'écarte de la forme du 'L'), le widget disparaît à nouveau. Si le geste arrive à la fin du 'L', le widget est tout à fait visible et se trouve sous le curseur. Il suffit alors de cliquer dessus pour l'activer.

Comme pour toute technique utilisant la reconnaissance de formes sans changement explicite de mode (besoin ou non de la reconnaissance), il est nécessaire de trouver un compromis entre simplicité de la forme d'appel du widget et risque d'appel involontaire. Néanmoins, plusieurs menus peuvent être disponibles en variant les formes d'appels (par exemple quatre 'L' superposés à nonante degrés d'intervalle).

3.2.5 Navigation sous contraintes

Dans une application 3D, l'opération la plus courante est l'inspection (peindre sur une surface, léger déplacement pour mieux percevoir les volumes, ...). Pour aider à cette tâche, [KKS+05] contraint la caméra à se déplacer le long de la surface de l'objet inspecté en gardant la caméra orientée le long de la normale de la surface.

Une autre contrainte fortement utilisée bien connue dans les jeux vidéos est celle qui consiste à ne pas pouvoir passer au travers du sol ou des murs. Elle n'est cependant utilisée que lorsqu'il faut simuler le déplacement à pied dans la scène.

3.2.6 Gestes simples et zones de l'écran

Dans [ZF99], les auteurs présentent une technique de choix d'opération de navigation 3D basée sur une notion de geste ramené à sa plus simple expression. La technique fait la différence entre un trait (pointe pressée sur l'affichage) qui commence horizontalement ou un trait qui commence verticalement. Dès que la distance limite pour décider de l'orientation est atteinte, le mouvement du stylo est continué et pilote un mouvement de caméra. Si le mouvement est initié horizontalement, l'opération est un panning (section 3.1.1) ; s'il est démarré verticalement, l'opération est soit un dollying (mouvement le long de l'axe perpendiculaire à l'écran) si le mouvement qui suit est vertical, soit un panning horizontal si le mouvement est horizontal.

Il faut remarquer que le mode est déterminé par le début du mouvement. Une fois le mode engagé, le mouvement peut être poursuivi avec un mélange d'horizontalité et de verticalité. En particulier, si l'on souhaite un panning vertical, le mouvement doit démarrer horizontalement ce qui introduira un léger décalage horizontal qui peut être corrigé naturellement dans la suite.

En basant le choix du mode sur la différence horizontal/vertical, seuls deux choix sont possibles. L'opération de rotation est réalisée en commençant le mouvement au bord de l'écran. Pour une rotation autour d'un point précis, il est possible de cliquer sur l'écran et ainsi de placer un centre de rotation.

3.2.7 Radial Scroll et Curve Dial

Radial Scroll[Smcs04] est un widget qui remplace le mécanisme de barre de défilement. Au lieu d'utiliser la barre pour faire défiler le contenu d'un document (par exemple une page web), l'utilisateur réalise des cercles avec le stylet posé sur la surface tactile. Tourner dans le sens des aiguilles d'une montre fait défiler vers le bas et tourner dans le sens inverse fait défiler vers le haut.

Radial Scroll possède un défaut : si l'utilisateur ne fait pas attention à tourner autour du centre du widget, le défilement ne se fait plus correctement. *Curve Dial*[SmcsB05] résout ce problème en supprimant la notion de centre : seule la courbure du mouvement réalisé par le stylet importe. De cette manière, l'utilisateur peut utiliser le widget sans maintenir les yeux fixés dessus.

Chapitre 4

Implémentation des widgets

Différents prototypes de widgets de navigation dans une scène, aussi bien 2D que 3D, ont été réalisés. Ils ont été implémentés avec la bibliothèque Java 3D. Ce chapitre commence par une présentation de la programmation 3D et de quelques concepts clés de Java 3D. La seconde moitié du chapitre concerne les différents prototypes de widgets qui ont été développés.

4.1 Java 3D

Les développements informatiques au Lucid Group se font essentiellement avec le langage de programmation Java[GJSB05]. Pour les développements nécessitant des rendus 3D, il existe plusieurs bibliothèques disponibles pour Java. Il existe notamment la bibliothèque JOGL qui permet d'utiliser la bibliothèque OpenGL dans un programme Java.

OpenGL (Open Graphics Library) [SWND05] est une spécification[1], originellement développée par la société SGI, qui spécifie une API pour écrire des applications 2D et 3D bénéficiant d'une accélération matérielle (c'est-à-dire que les instructions offertes par l'API sont exécutées par un processeur graphique spécifique plutôt que par le CPU de l'ordinateur). Avec OpenGL, il est possible d'utiliser les capacités offertes par le matériel de façon uniforme, indépendamment de sa marque ou de son modèle. La bibliothèque implémentant cette API est généralement conçue par la firme fournissant le matériel. L'alternative courante à OpenGL est Direct3D, de

[1] OpenGL est bien une spécification plutôt qu'une bibliothèque de programmation. Néanmoins nous nous permettons d'employer le terme de bibliothèque OpenGL pour désigner de façon générique toute bibliothèque qui implémente cette spécification.

Microsoft. Elle est disponible uniquement sous Windows alors qu'OpenGL est disponible entre autre sous Windows, Mac OS X et Linux. La portabilité des applications développées au Lucid Group impose donc d'utiliser OpenGL.

D'autres bibliothèques 3D, construites sur JOGL, sont disponibles. La bibliothèque choisie au Lucid Group pour les applications 3D est Java 3D[SRD00]. Une des raisons d'un tel choix est le concept de "write once, run everywhere" de Java et l'application de ce concept à Java 3D. Java 3D peut utiliser Direct3D ou OpenGL sous Windows et utilise OpenGL sous Linux et Mac OS. Il est également possible d'écrire une application avec Java 3D pour qu'elle fonctionne dans un navigateur web.

Java 3D a été initialement développée par Sun Microsystems. Il s'agit d'une bibliothèque de haut niveau (contrairement à OpenGL ou JOGL) où l'organisation et la gestion d'une application 3D se fait au travers du concept central de graphe de scène (ou *scenegraph* en anglais). Java 3D permet de créer, modifier dynamiquement et rendre un tel graphe.

4.1.1 3D bas niveau et graphe de scène

Une bibliothèque de programmation 3D de bas niveau comme OpenGL expose des commandes exécutées par l'ordinateur. Ces commandes sont de deux types. Certaines commandes permettent de manipuler l'état du système graphique et d'autres permettent d'émettre des primitives 3D. L'état du système comprend la couleur courante, la couleur à utiliser lorsque l'écran est vidé, la taille courante des points ou des bords de polygones à dessiner, les lampes actives, la texture active, etc. Les commandes d'émission de primitives permettent de dessiner à l'écran des points, des lignes et des polygones. Le rendu de ces primitives est dépendant de l'état du système.

Cette manière d'opérer est décrite comme 'directe'. Le programmeur doit procéder instruction par instruction dans un ordre précis pour obtenir le résultat désiré. Ceci est compréhensible si l'on considère qu'OpenGL a été pensé comme une bibliothèque pour le langage C, langage impératif. Par opposition au terme 'directe', il existe des bibliothèques de plus haut niveau dites en 'mode retenu' (en anglais, *retained mode*). Au lieu d'indiquer impérativement au système graphique ce qu'il doit faire, le programmeur manipule une structure de données. Typiquement, une telle structure sera un graphe de scène qui abstrait les commandes de bas niveau sous-jacentes. Au lieu d'écrire des suites de commandes, le programmeur manipule des objets (au sens de la programmation orienté-objet).

L'API Java 3D est composée d'un ensemble principal de classes basées sur la notion de graphe de scène et de classes implémentant les bases d'algèbre linéaire

nécessaires à la programmation 3D (concepts de points, de vecteurs, de matrices de transformation, ...). Dans Java 3D, les différentes composantes d'une scène sont organisées en arbre. Les différents noeuds dans l'arbre peuvent correspondre à la géométrie des objets et à leur placement les uns par rapport aux autres ainsi qu'à des indications sur la façon de réaliser le rendu (éclairage, textures, ...). Le graphe de scène supporte également la détection de collision entre objets 3D et le picking. Un exemple de graphe de scène est illustré à la figure 4.1.

4.1.2 Noeuds principaux

Une bibliothèque basée sur la notion de graphe permet d'en construire un et prend en charge le rendu. Celui-ci se fait en traversant le graphe et en modifiant l'état de la bibliothèque 3D de bas niveau sous-jacente. Au fur et à mesure que le graphe est traversé, deux choses se passent qui correspondent à l'état du système graphique et à l'émission des primitives. D'une part, l'état du système est modifié en fonction des noeuds d'état (lumière, placement des objets, ...). D'autre part, lorsqu'un noeud représentant un objet graphique est atteint, les primitives permettant d'afficher cet objet sont émises en utilisant la bibliothèque de bas niveau.

Chaque noeud dans le graphe est une instance d'une classe Java 3D (ou une implémentation d'une interface Java 3D). Il y a deux types de noeuds principaux : le groupe et la feuille. Les feuilles sont des noeuds qui ne possèdent pas de sous-noeuds dans le graphe. Elles peuvent représenter la géométrie d'un objet (comme une sphère ou un bâtiment), l'apparence d'un tel objet (comme sa couleur) ou bien caractériser l'environnement (comme l'éclairage ou le son). Un noeud caractérisant la géométrie est associé à un noeud de type 'apparence'. Ils forment ensemble un noeud 'forme'.

Le deuxième type principal de noeud est le 'groupe'. Comme son nom l'indique, ce noeud permet de grouper plusieurs noeuds (ses enfants). Il y en a de différentes sortes. Certains ne servent qu'à créer un groupe de noeuds pour les manipuler facilement. D'autres permettent d'afficher les noeuds 'enfants' dans un certain ordre (pour réaliser des effets particuliers, notamment le chevauchement d'objets transparents) ou encore de pouvoir les placer à un endroit précis de la scène.

En plus des caractéristiques liées au rendu, chaque noeud en possède d'autres qui décrivent l'intention du programmeur. Par exemple, on peut décider si un noeud groupe peut recevoir de nouveaux enfants à l'exécution du programme ou non. Ce genre d'information permet à Java 3D de modifier le graphe de la scène pour pouvoir le gérer le plus efficacement possible. Cette modification est appelée *compilation*.

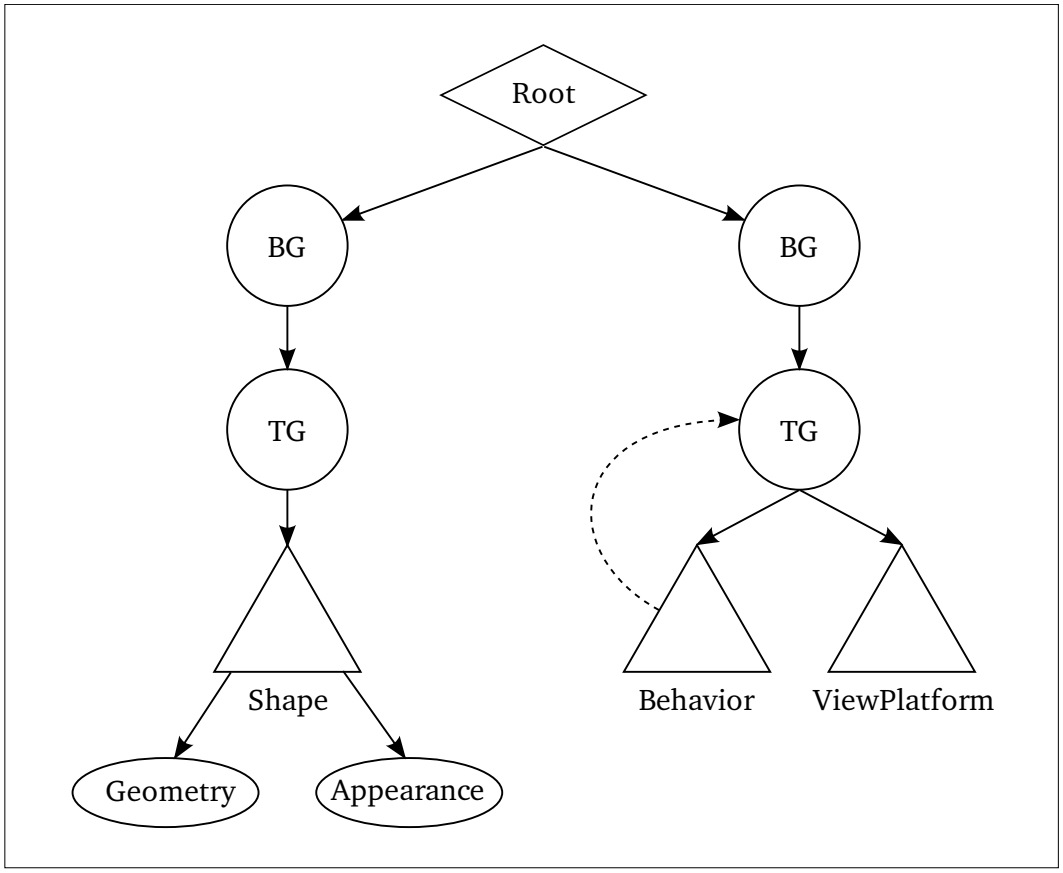


FIG. 4.1 – *Exemple de graphe de scène. Le graphe débute au noeud racine (Root) et possède deux branches. Chaque branche commence par un noeud BranchGroup (BG). La branche de gauche contient un noeud TransformGroup (TG) qui positionne dans la scène une forme (Shape). Cette forme est composée d'une géométrie et d'une apparence. Par exemple, si la forme représente un cube rouge, la géométrie comprendra les six faces du cube et l'apparence spécifiera une couleur rouge. La branche de droite possède également un noeud TransformGroup. Ce noeud place dans la scène la plate-forme de visualisation (on peut la considérer comme la caméra virtuelle). Il place également un noeud Behavior qui contrôle (flèche en pointillés) la transformation au-dessus d'eux.*

Plate-forme de visualisation

Parmi les noeuds feuilles, il y en a un de particulier : la plate-forme de visualisation (la classe *ViewPlatform*). Ce noeud correspond à la caméra virtuelle placée dans la scène comme n'importe quel autre noeud Java 3D. Il peut servir aussi de point d'attache pour d'autres noeuds qui doivent être positionnés par rapport à la caméra.

Noeuds de type 'groupe' et transformation

Pour organiser les objets de la scène dans le graphe et les placer où il le souhaite, le programmeur fait essentiellement usage de deux types de noeuds 'groupe'. Il s'agit des noeuds *BranchGroup* et *TransformGroup* (d'après les noms des classes Java qui y correspondent).

Un noeud BranchGroup est une sous-classe de Group. Il permet de définir une sous-branche du graphe qui peut être attachée et détachée du reste du graphe à l'exécution. Il expose une méthode qui permet de compiler le sous-graphe qu'il définit.

Un TransformGroup est un noeud de type 'groupe' qui contient un objet de la classe *Transform3D* (qui représente une tranformation 3D), c'est-à-dire que le TransformGroup positionne, oriente et donne l'échelle de ses enfants. C'est l'utilisation du noeud TransformGroup qui donne tout son sens à l'organisation en graphe des éléments qui composent la scène[2]. La tranformation d'un objet dans la scène est donnée par la concaténation des transformations traversées dans le chemin direct de la racine du graphe jusqu'au noeud représentant l'objet. La transformation résultant de la concaténation transforme les points et les normales de l'objet en coordonnées globales.

Un objet de la classe Transform3D spécifie une transformation applicable à des noeuds du graphe de la scène. Cette transformation est représentée dans la classe Transform3D par une matrice 4x4 de nombres à virgule flottante à double précision. La classe Transform3D permet de créer des transformations particulières, par exemple une rotation sur l'axe des x ou une rotation d'après la valeur d'un quaternion.

[2]En plus d'établir le positionnement hiérarchique des éléments de la scène, le graphe permet d'éliminer efficacement du rendu des portions complètes de la scène qui ne seraient de toute façon pas visibles à l'écran.

4.1.3 Déplacement de la caméra

Le principe pour implémenter la navigation dans une scène Java 3D est de manipuler le noeud TransformGroup qui se trouve au-dessus de la ViewPlatform dans le graphe, c'est-à-dire le noeud qui positionne la caméra virtuelle par laquelle l'utilisateur voit la scène. Les manipulations que nous désirons réaliser sont pour la plupart liées aux mouvements du dispositif de pointage. Par exemple, pour déplacer la caméra au-dessus de la feuille virtuelle dans une application 2D, un déplacement de n pixels du curseur doit déplacer la caméra, en coordonnées de l'écran, de n pixels (dans la direction opposée comme expliqué précédemment (section 2.1.1), si l'on veut que le curseur soit fixe par rapport à la feuille).

Réaction aux évènements

Pour réagir aux mouvements du curseur (ou à d'autres évènements d'entrée), Java 3D propose un mécanisme à travers la classe *Behavior*.

Le programmeur doit sous-classer la classe Behavior et implémenter deux méthodes. La première permet d'initialiser le nouveau Behavior et la seconde est exécutée par Java 3D chaque fois qu'un évènement, appelé 'stimulus' dans le cadre de la classe Behavior, survient. Le lecteur curieux se demande probablement pourquoi un objet de la classe Behavior est également un noeud prenant place dans le graphe de la scène. La raison en est simple. A un Behavior est associé un volume d'activation. Si la position de la caméra virtuelle ne se trouve pas dans le volume du Behavior, alors le Behavior ne recevra pas de stimuli. Ce principe de volume d'activation est également utilisé pour les noeuds associés à des sons ou à des éclairages et permet d'éviter que l'ordinateur fasse des calculs dont les résultats ne seraient pas visibles.

Dans le cas présent, étant donné les contraintes que l'on s'est fixé, les seuls stimuli qui nous intéressent sont les évènements de la bibliothèque de fenêtrage AWT[3] qui concernent la souris. En effet, peu importe le périphérique de pointage utilisé, les évènements sont conceptuellement associés à ceux d'une souris. AWT est la bibliothèque standard pour réaliser des interfaces graphiques avec le langage de programmation Java. Pour une application Java 3D, AWT sert à obtenir une fenêtre dans laquelle la scène est rendue ainsi que les évènements qui y sont liés, comme le changement de taille de la fenêtre ou son recouvrement/découvrement par une autre fenêtre à l'écran. Nous ne sommes pas concerné par ces aspects et utilisons les mots 'fenêtre' et 'écran' comme s'ils étaient interchangeables.

[3]AWT est une bibliothèque permettant de créer des fenêtres à l'écran et de gérer les évènements en entrée.

4.1.4 Affichage 2D

Le widget est affiché comme un objet 2D mais est programmé avec Java 3D. Cela permet si nécessaire de lui donner un aspect différent, y compris de l'animer en trois dimensions, et de mieux l'intégrer au reste de l'application. Dans les premières ébauches, même son placement se faisait dans la scène 3D plutôt qu'en 2D en surimpression de l'affichage de la scène[4].

Problème

Contrairement à ce que l'on pourrait attendre d'une telle bibliothèque, l'affichage en 2D est rendu difficile avec Java 3D car l'API ne permet pas de passer d'une projection perspective à une projection orthogonale dans le même rendu. Une projection orthogonale ne rend pas les éléments de la scène de plus en plus petits en fonction de leur éloignement par rapport à la caméra. Elle est utile par exemple pour réaliser des plans. Elle est également utile pour l'affichage en deux dimensions car elle permet une correspondance naturelle entre les unités utilisées pour placer l'objet 2D dans la scène et les pixels de la surface d'affichage (par exemple, si l'objet est un rectangle sur lequel est placé une image de 20x10 pixels, nous aimerions qu'il occupe 20x10 pixels à l'écran).

Solution

La solution retenue est de placer les éléments 2D dans la scène de façon à ce que leur affichage fasse correspondre un pixel de l'élément à rendre avec un pixel de l'écran (pour éviter tout clignotement lors de l'animation) en plaçant des noeuds avec les transformations adéquates dans le graphe de scène. Ces transformations changent en même temps que le placement de la caméra pour que les objets 2D restent fixes par rapport à celle-ci (voir figure 4.2).

Inconvénient

L'inconvénient de cette méthode, puisque l'objet 2D est finalement un objet parmi d'autres dans la scène complète, réside dans la possible de créer une situation où un objet de la scène (un mur par exemple) se situe entre la caméra et l'élément 2D, le cachant ainsi à la vue. Ce problème peut être minimisé en prenant soin d'afficher

[4]Ce placement en trois dimensions rendait aisé la détermination du centre de rotation de la caméra par exemple.

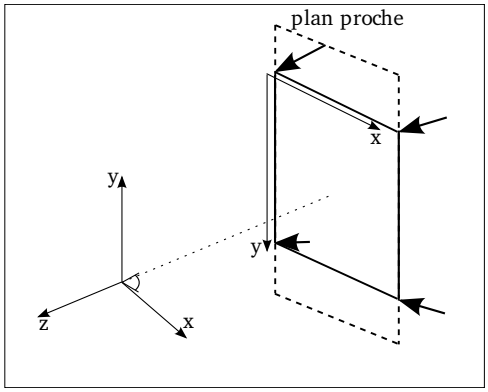


FIG. 4.2 – *La caméra définit un système de coordonnées dont elle est le centre. Elle est pointée le long des z négatifs. L'axe des x va vers la droite et l'axe des y va vers le haut. Le système de coordonnées de l'écran possède deux axes : l'axe des x va vers la droite et l'axe des y va vers le bas. Le point (0,0) est situé en haut à gauche.*

les éléments 2D très proches de la caméra (c'est-à-dire le plus près possible du plan proche, à l'intérieur du volume de visualisation). Ou encore, il pourrait être totalement supprimé en s'assurant que le rendu des éléments 2D se fasse sans tester la profondeur relative[5] à l'écran des éléments 2D par rapport à la scène. Malheureusement, cette possibilité ne s'est pas avérée satisfaisante avec Java 3D.

Calcul de la transformation

Pour placer un objet 2D mesuré en pixels dans la scène 3D de façon à ce que chaque pixel de l'objet 2D corresponde à un pixel de l'écran, il faut connaître la transformation utilisée pour projeter un objet de la scène à l'écran. Une fois la transformation connue, le placement au pixel près est assuré par un noeud contenant la transformation inverse. La transformation inverse permet de passer des coordonnées écran aux coordonnées de la scène avant projection. Il est à noter que le système de coordonnées avant projection n'est pas ce système de coordonnées global. Dans le système de coordonnées avant projection, le centre du repère est donné par la position de la caméra. Autrement dit, la caméra est placée en (0,0,0) dans ce système de coordonnées. Le système est orienté de telle manière que la caméra "regarde" le long des z négatifs. Le choix d'un tel système repose sur les simplifications qu'il

[5]Lorsqu'un objet est affiché par le système graphique, la formation des pixels correspondant à cet objet doivent passer un test de profondeur. Ce test vérifie qu'il n'y a pas déjà eu des pixels au même endroit de l'écran en provenance d'un objet plus proche. Si le test n'est pas passé avec succès, les nouveaux pixels ne sont pas affichés.

apporte à l'implémentation de la projection perspective[6].

Tout objet 2D à placer précisément à l'écran est décrit en coordonnées de l'écran (en utilisant le pixel comme unité) et est inséré dans le graphe de scène sous ce noeud. La transformation totale appliquée à l'objet le place dans la scène, en face de la caméra, sur le plan proche et les coordonnées (0,0) de l'écran se trouvent sur le coin supérieur gauche de la face du volume de visualisation. Ensuite, la projection de la scène assurée par Java 3D projette cet espace sur celui de l'écran. Dans le cas présent, la transformation de projection est une transformation perspective.

Le frustum utilisé en projection perspective n'est pas nécessairement symétrique mais celui utilisé normalement par Java 3D l'est et est simple à reproduire. Mais nous ne cherchons pas à reproduire l'intégralité de la transformation de projection ; la transformation qui fait correspondre la base du frustum à l'écran nous suffit. Connaissant l'angle du champ de vision utilisé par Java 3D[7] et la distance du plan proche, un peu de trigonométrie permet de calculer les dimensions (hauteur et largeur) de la base du frustum (fig. 4.3). Ces valeurs, avec la résolution de l'écran, permettent de calculer le changement d'échelle entre les deux rectangles. Pour compléter la transformation, il reste à effectuer une translation le long de l'axe des z (en direction des z négatifs) d'une valeur égale à la profondeur du plan proche et une translation parallèle à l'écran de la moitié de ses dimensions pour centrer le plan.

Le code complet pour réaliser la transformation est illustré à la figure 4.4. Deux détails sont à noter. Le premier est le signe moins à la ligne `-2.0 / height,`. Son effet est d'inverser l'axe y car en coordonnées de l'écran, cet axe va du haut vers le bas alors que dans le système de coordonnées 3D utilisé par Java 3D, il croît vers le haut. Le second détail est l'ajout des valeurs 0.375 sur les deux axes lors de la translation parallèle à l'écran. Ces valeurs sont choisies pour assurer le placement au pixel près. Les calculs de projections se font avec des nombres à virgules flottantes et sans l'ajout de cette constante, en fonction du placement de la caméra (et donc du plan 2D que l'on vient de placer) dans la scène, un pixel du plan 2D ne serait pas toujours "en face" du même pixel de l'écran, causant un scintillement lors d'un mouvement de caméra[SWND05].

---

[6]En particulier, la matrice de transformation correspondant à la projection perspective peut être ramenée à une division par la valeur en z des coordonnées à projeter. A nouveau, on se rend compte que plus un objet sera éloigné (la valeur absolue en z est importante) plus il sera réduit à la projection.

[7]L'angle de champ de vision capture l'axe des x (la commande OpenGL gluPerspective utilise un angle sur l'axe vertical) et est disponible à partir de la classe *View*, liée à la classe *Canvas3D*.

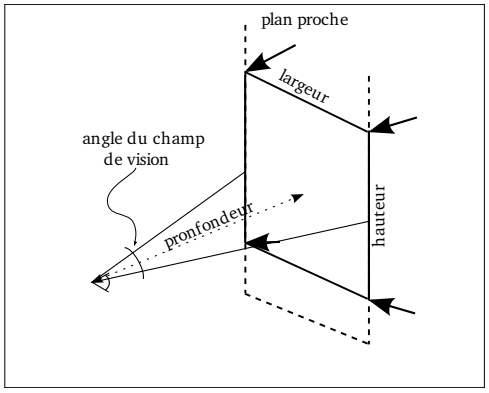


FIG. 4.3 – *A partir de la profondeur du plan proche et de l'angle de champ de vision, on peut calculer la largeur et la hauteur de la base du frustum.*

4.2 Prototypes de widgets de navigation

Chacun des prototypes de widgets de navigation est réalisé comme un menu affiché en deux dimensions par dessus la scène pouvant être placé où l'utilisateur le souhaite. Dans la première partie de cette section, nous envisageons les différentes manières de déplacer la caméra. Dans la seconde partie, nous présentons les différents widgets qui permettent de piloter la caméra.

Il y a beaucoup de possibilités et de choix à faire concernant l'interaction attendue de l'utilisateur pour contrôler la caméra. Lors d'une réunion pendant le stage, nous avons présenté à l'équipe des techniques d'interactions à partir de la littérature scientifique. Les choix des types des widgets ont ensuite été faits avec l'équipe sur base de cette présentation. D'autres choix étaient des idées présentes dans l'équipe avant le début du stage. D'autres choix encore ont été faits au cours du stage.

4.2.1 Usage d'une opération

Pour réaliser un mouvement de la caméra, il faut deux étapes : changer de mode en indiquant l'opération désirée (une translation, une rotation, ...) et utiliser cette opération.

```
canvasWidth  = canvas.getWidth();
canvasHeight = canvas.getHeight();
double fov   = canvas.getView().getFieldOfView();
double invAspect = canvasHeight / canvasWidth;

xmax = depth * Math.tan( fov / 2.0 );
xmin = -xmax;
ymax = xmax * invAspect;
ymin = -ymax;
width  = xmax - xmin;
height = ymax - ymin;

Transform3D t0 = new Transform3D();
Transform3D t1 = new Transform3D();
Transform3D t2 = new Transform3D();
t0.set( new Vector3d(
        0.375,
        0.375,
        0.0
    ) );
t1.setIdentity();
t1.setScale( new Vector3d(
             width  / canvasWidth,
            -height / canvasHeight,
             1.0
         ) );
t2.set( new Vector3d(
        xmin,
        ymax,
       -depth
    ) );

t2.mul( t1 );
t2.mul( t0 );
```

FIG. 4.4 – *Code pour calculer la transformation qui passe des coordonnées écran aux coordonnées de la plate-forme de visualisation.* `canvasWidth` *et* `canvasHeight` *sont la largeur et la hauteur de l'écran.* `width` *et* `height` *sont la largeur et la hauteur de la base du frustum.* `fov` *est le field of view : l'angle qui sous-tend horizontalement le frustum.* `depth` *est la distance entre le point de vue et le plan proche. A la fin du code,* `t2` *contient la transformation voulue.*

Activation - usage - désactivation

Le principe le plus courant est l'activation d'un mode en cliquant sur le bouton correspondant, son utilisation, puis sa désactivation en cliquant à nouveau sur le même bouton ou en sélectionnant un autre mode. C'est par exemple le principe utilisé par EsQUIsE.

Activation et usage

Un principe similaire qui ne demande pas de cliquer à deux reprises (en supposant que l'on désire revenir au mode de base après l'opération) consiste à passer dans le mode de l'opération dès la pression du stylet sur un bouton et à revenir au mode de base dès que le stylet quitte la surface de travail. Pour utiliser le mode de l'opération, il faut déplacer le stylet entre ces deux étapes.

Activation - usage unique

Entre les deux principes précédents, il est possible de réaliser un compromis. L'activation du mode de l'opération se fait par un click sur un bouton. Utiliser l'opération se fait avec le stylet sur la surface. Le fait de quitter la surface ensuite désactive le mode.

Combinaison

Le premier principe utilise un click sur un bouton et le second principe utilise un "glisser" sur le bouton (c'est-à-dire presser le stylet puis de le déplacer sans le relever). Ces deux types d'évènements peuvent être distingués par l'application et il est donc possible de réaliser un widget dont les boutons offrent les deux principes.

Usage automatique

Dans les principes précédents, la pression du stylet ou le click permettent d'activer le mode correspondant à l'opération mais celle-ci est finalement contrôlée par le déplacement du stylet. Rien n'empêche que la pression du stylet sur un bouton fasse mouvoir la caméra à une certaine vitesse jusqu'au moment ou le stylet quitte la surface, sans nécessiter de mouvement du stylet.

Cette façon de procéder est celle utilisée dans les jeux vidéos : tant que la touche “haut” est pressée, le personnage marche en avant. Il s’arrête dès que la touche n’est plus enfoncée.

Dans la description qui précède, le mouvement du curseur n’a plus de fonction mais on peut lui en retrouver une : le mouvement a lieu ou non suivant que le bouton est pressé ou non et la vitesse du mouvement est contrôlée par la distance entre le curseur et le bouton. Lorsque le bouton est enfoncé, le curseur est positionné dessus et la vitesse est minimale. Enfin, la position du curseur peut également contrôler le mouvement, par exemple en donnant la direction d’une rotation.

4.2.2 Opérations

Les opérations sont de trois types : translation, rotation et zoom (changement d’échelle). Pour déclencher le mouvement de la caméra, puis réaliser le mouvement proprement dit et enfin le terminer, il y a plusieurs possibilités. Celles-ci sont exposées à la fois dans le cas de la navigation 2D et de la navigation 3D. Ensuite, nous présentons la façon dont les opérations s’inscrivent dans Java 3D.

Translation

Par translation, nous parlons essentiellement de *panning*, c’est-à-dire une translation qui se fait parallèlement à l’écran (ou au plan proche si l’on préfère). Le panning est donc une opération à deux degrés de libertés et il peut être réalisé en faisant correspondre les deux dimensions du périphérique de pointage aux deux dimensions de la translation de la caméra. Le curseur peut se déplacer dans le sens de la caméra ou dans le sens contraire (suivant la métaphore *scene-in-hand* ou *eyeball-in-hand*). Le mouvement de caméra peut dépendre ou non du facteur de zoom courant. S’il dépend du facteur de zoom, un cas particulier se présente où le curseur est fixe par rapport à la scène pendant le déplacement (à condition que le curseur se déplace dans le sens contraire de la caméra).

En 3D, le panning est similaire au cas 2D sauf si une projection perspective est employée. Comme celle-ci modifie la taille d’un objet en fonction de sa distance par rapport au point de vue, il n’est pas possible de parler d’un curseur fixe par rapport à la scène. Par contre, il peut être fixe par rapport à un point le long du rayon de *picking* (par définition, tous les points le long de ce rayon sont projetés au même endroit à l’écran).

Garder un point fixe sous le curseur est également possible si la translation n’est pas parallèle à l’écran. Par exemple, la translation peut être effectuée parallèlement à un plan horizontal.

Zoom

Le zoom ne possède qu'un degré de liberté. On peut alors faire correspondre ce paramètre au périphérique de pointage de différentes manières. Zoomer peut se faire en déplaçant le stylet horizontalement et/ou verticalement. Il est également possible d'utiliser une technique comme *Curve Dial*.

A la section 2.1.1, l'effet de la position de référence est montré pour un zoom en 2D. En 3D, le zoom est généralement un déplacement sur l'axe z. A la place, on peut tenir compte de l'emplacement du curseur et faire déplacer la caméra le long du rayon de picking.

Rotation

La rotation 2D est similaire au zoom dans le fait qu'elle ne possède qu'un seul degré de liberté. Les possibilités de correspondance entre mouvements de caméra et mouvements du stylet sont également similaires mais il y a une correspondance supplémentaire : étant donné le centre de rotation, il est possible de déterminer l'angle de rotation par rapport à l'angle formé par le mouvement du curseur entre son point initial et son point courant. Le curseur n'est pas situé en permanence sur le même point de la scène mais sur une même ligne tout au long de l'opération.

La rotation dans une scène 3D possède trois degrés de liberté, un pour chaque axe. C'est également le cas pour la translation mais celle-ci en traite deux avec le panning et le troisième avec le zoom. Mais seuls deux axes nous intéressent. En effet, la rotation autour de l'axe des z donne l'impression de pencher la tête sur le côté et ce point de vue n'est pas fort utile pour visualiser des bâtiments. De cette manière, la correspondance entre le mouvement de la caméra et le périphérique de pointage est semblable au cas du panning.

La différence entre les métaphores *scene-in-hand* et *eyeball-in-hand* dépend du sens du mouvement de la caméra par rapport à celui du curseur. Pour la rotation en trois dimensions, la différence tient à la position du centre de rotation. S'il coïncide avec la position de la caméra, tourner celle-ci donne l'impression de tourner la tête. Autrement, cela nous fait orbiter autour de la scène.

Navigation et Java 3D

Les opérations les plus simples à implémenter sont les translations de la caméra dans un plan parallèle à l'écran et la translation le long d'un axe perpendiculaire à l'écran (les axes locaux de la caméra). Le noeud de transformation (instance de la classe TransformGroup) du graphe de scène sous lequel se trouve la caméra positionne celle-ci dans la scène. Il suffit de composer la transformation contenue dans ce noeud avec une transformation représentant une translation sur l'axe des X, Y ou Z, pour réaliser une translation de la caméra sur ces axes. Java 3D permet de créer une transformation de translation à partir d'un vecteur (indiquant à la fois la direction et l'amplitude du mouvement). Généraliser la translation de la caméra à d'autres directions ne pose donc aucun problème.

Pour spécifier complètement le mouvement, il est nécessaire de connaître, en plus de la direction, son amplitude. Plusieurs possibilités s'offrent à nous pour déterminer l'amplitude. Pour rendre la manipulation la plus intuitive possible, il est pratique de faire correspondre le mouvement de telle manière qu'un point de la scène reste sous le curseur pendant toute la manipulation. Cela donne l'impression à l'utilisateur de pouvoir déplacer la scène relativement à la caméra en "l'épinglant" avec le curseur.

Un cas particulier de translation de la caméra est celui qui se fait parallèlement à un plan horizontal (que l'on peut considérer comme le sol ou comme un plan de référence sur lequel la scène est construite). A condition que la ligne de vision ne soit pas proche d'une parallèle avec ce plan, le plan peut servir de repère pour une manipulation directe. Le principe de la translation par manipulation directe d'un plan semble approprié lorsque la signification du plan est bien perçu par l'utilisateur. Ce type de translation permet de se déplacer, sans changement d'orientation de la caméra, intuitivement et rapidement sur de grands espaces.

Calculer l'amplitude du mouvement revient à calculer, à un facteur près, la distance entre la position du curseur au début de l'opération et sa position au moment courant.

Les opérations de rotation sont réalisées en multipliant plusieurs transformations. Chacune de ces transformations réalise une des étapes décrites à la section 2.1.1.

4.3 Changements de mode

Le problème du changement efficace de mode d'opération de navigation a été travaillé dans le cas d'une scène en deux dimensions. Le cas de la scène à trois dimensions sert à explorer quelles sont les opérations de navigation les plus simples

à utiliser. Les deux aspects ont été séparés pour ne pas accroître les combinaisons opérations/changement de mode. Une bonne solution de changement de mode dans le cas 2D devra être légèrement modifiée pour tenir compte des degrés de liberté supplémentaires dans le cas 3D.

Les solutions explorées pour invoquer les opérations sont toutes à base d'un menu flottant par dessus la scène. La position du menu donne un point de référence pour la rotation et la mise à l'échelle simple à appréhender (notion d'*affordability* en anglais) et à déplacer (en déplaçant le menu). Nous avons choisi d'utiliser un menu principalement pour cette raison et parce que l'opération de base est le dessin. De ce fait, l'utilisation de la reconnaissance de geste est plus difficile car il faut discriminer entre la volonté de changer d'opération et celle de dessiner. Les types principaux des widgets développés sont illustrés à la figure 4.5.

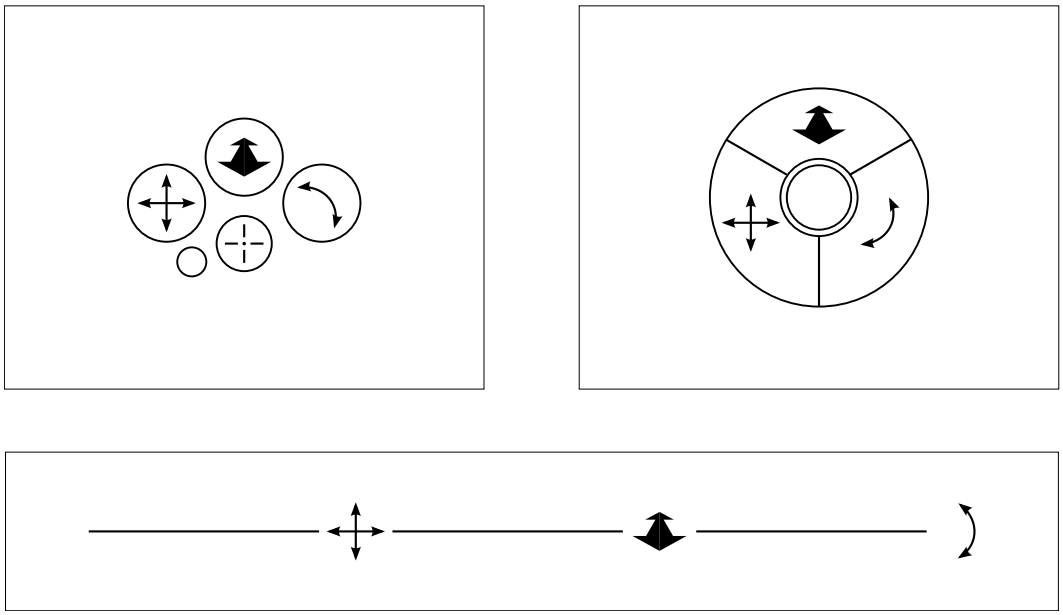


FIG. 4.5 – *Le widget en haut à gauche fournit trois boutons, un pour chaque mode. Le bouton du bas sert à déplacer le widget et correspond au point de référence pour la rotation et la mise à l'échelle. Le plus petit bouton permet de réinitialiser la vue. Le widget en haut à droite est le plus proche du marking menu. Lorsqu'il n'est pas utilisé, seul le cercle central est visible. Une fois appelé, il se positionne à l'emplacement du curseur. Un trait dans une des trois directions sélectionne une opération et le widget disparaît. L'emplacement de l'appel définit le point de référence. Le widget du bas utilise le principe des croisements sans pression du stylet.*

La présence du menu permet de séparer l'utilisation du mode courant (par exemple le dessin ou la translation) et le passage à un autre mode. Comme le mode de dessin est le mode principal, il était prévu initialement que les autres modes soient transitoires. Cela signifie qu'après avoir été invoqués et utilisés, le mode dessin redevient le mode courant. Toujours initialement, le changement de mode et l'utilisation d'un mode étaient regroupés : plutôt que de cliquer sur un bouton correspondant au mode translation puis de faire un presser-déplacer pour effectuer la translation et enfin revenir au mode dessin au relâchement, le mode est entré au moment de la pression du bouton, utilisé pendant le glissement et quitté au moment de relâcher le bouton (voir "Activation et usage" dans la section 4.2.1).

Une autre raison avantageuse d'utiliser un menu flottant est qu'un tel menu est manipulé comme un objet placé sur la scène bien que le contexte actuel est celui de la navigation et pas la manipulation ou l'édition d'objets. Néanmoins, dans le cas de la navigation par manipulation directe de la scène, les deux types d'interactions sont semblables et l'utilisateur ne doit se familiariser qu'avec un seul concept.

4.3.1 Boutons

Le premier prototype réalisé utilise des boutons suivant les principes décrits plus haut. Ensuite, les autres variations de la section 4.2.1 ont été faites.

4.3.2 Barres à croiser

Dans ces prototypes, le principe est de remplacer les trois boutons du menu flottant de base et de proposer à la place trois traits horizontaux placés côte-à-côte. Pour utiliser ce widget, il suffit de faire passer le curseur à travers l'un des traits sans pression sur la surface de travail. Pour que cela soit possible, il faut que le périphérique de pointage puisse localiser le stylet sans qu'il y ait contact entre sa pointe et la surface tactile. Techniquement, du point de vue de la bibliothèque AWT, cela revient à bouger la souris sans pression sur l'un de ses boutons. Les variations consistent à permettre l'activation des modes lorsque le curseur se présente uniquement par un côté précis de la barre ou lorsqu'il effectue plusieurs croisements en succession rapide.

Le point qui marque le début de l'utilisation d'une opération (là où le stylet commence à toucher la surface) définit le centre de référence pour les opérations de rotation et de mise à l'échelle.

4.3.3 Directions

Le principe des marking menus est utilisé pour un prototype qui est composé d'un seul bouton. Une marque démarrant dans ce bouton sélectionne une opération suivant sa direction et son sens. Sur les quatre points cardinaux, seuls trois sont utilisés.

4.3.4 Variations et appel du widget

Les prototypes à base de boutons utilisent les différents principes de la section 4.2.1. Il est bien entendu possible de combiner certains de ces principes avec les autres types de prototypes.

En plus des modes correspondants aux opérations de navigation, nous avons implémenté un mécanisme simple permettant de faire venir le widget à l'endroit souhaité sur le Bureau Virtuel sans manipuler directement le widget. Ce mécanisme consiste à presser la pointe du stylet sans le déplacer pendant un court laps de temps (300 millisecondes). Si le stylet n'a pas été bougé au bout de ce laps de temps, le widget change de position et apparaît à l'endroit du curseur. Il faut déterminer expérimentalement le meilleur compromis pour la durée du laps de temps. S'il est trop long, le widget manque de réactivité tandis que s'il est trop court, l'utilisateur risque de faire venir le widget alors qu'il souhaitait dessiner.

Enfin, un widget réunissant l'appel après un court délai, le changement de mode par un trait qui démarre dans une zone unique et le principe d'*Activation et usage*. Le widget n'est pas visible en temps normal. Il le devient lorsque l'utilisateur presse le stylet et attend un court instant. Une fois qu'il est visible, l'utilisateur, sans relever le stylet, forme un trait. La direction de ce dernier détermine le mode qui est entré et le widget disparaît. Toujours sans le relever, l'utilisateur déplace le stylet pour faire usage de l'opération. Le fait de relever le stylet quitte le mode de l'opération. Alors que les autres widgets possèdent un point de référence pour la rotation et le zoom (déterminé par picking dans le cas 3D), ce widget détermine le point de référence par l'emplacement du curseur lorsqu'il est pressé sur la surface.

Cette manière de déterminer le point de référence, par le lieu de départ d'utilisation d'une opération, crée une nouvelle série de variantes des widgets précédents.

Chapitre 5

Evaluation

Dans le chapitre précédent, nous avons exposé une série de types et de variations de widgets destinés à permettre à l'utilisateur de positionner un point de vue dans une scène, qu'elle soit en deux ou en trois dimensions. Pendant l'implémentation des widgets, nous avons eu l'occasion de faire tester certains de ceux-ci par des membres de l'équipe du Lucid Group. Un widget a également pu être utilisé par des étudiants en architecture utilisant le logiciel SketSha sur le Bureau Virtuel.

Nous n'avons pas procédé à une évaluation plus systématique et formelle des différentes variantes par manque de temps mais nous avons apporté des modifications à SketSha dans le but de réaliser une telle évaluation.

5.1 Opération à usage unique

Le principe d'usage unique d'une opération (dès que le stylet quitte la surface, l'application retourne au mode de base) évite à l'utilisateur de sélectionner explicitement l'opération de base, celle qui est employée en temps normal.

Lorsque l'usage unique est en plus couplé à l'activation du mode de l'opération (l'utilisateur presse le bouton lié à l'opération et le mouvement qui suit, sans relever le stylet, réalise l'opération sélectionnée), il est possible de créer une situation potentiellement gênante. Cette situation survient lorsque le widget est positionné près du bord du Bureau Virtuel et que le mouvement de stylet nécessaire à l'usage de l'opération va en direction de ce même bord. Puisque le stylet démarre près du bord, il touchera le bord rapidement et le mouvement du stylet est donc particulièrement limité. De ce fait, l'utilisateur devra à de multiples reprises activer, en positionnant le stylet dans la partie appropriée du widget, et utiliser l'opération.

Nous ne savons pas encore si cette situation se présente régulièrement et si elle est véritablement gênante pour l'utilisateur.

5.2 Barres à croiser

Le widget est composé de segments de droites à la place de boutons. Croiser une barre avec le curseur, sans pression sur le Bureau Virtuel, active le mode correspondant. A l'utilisation au Bureau Virtuel, les personnes qui l'ont essayé l'ont d'abord trouvé désagréable mais après quelques minutes, l'impression laissée était beaucoup plus positive. Ceci est vrai à condition d'être assis au bureau car faire survoler le stylet assez près de la surface est un mouvement assez naturel dans ce cas. En position debout, maintenir le stylet suffisamment proche de la surface sans l'y appuyer demande un effort de la part de l'utilisateur. Cet effort ne correspond pas à la visée initiale d'utilisation "sans y penser" du bureau. En effet, il n'est pas anormal d'utiliser le Bureau Virtuel sans s'asseoir au préalable.

Lorsque l'utilisateur est assis à la table, le mouvement suffisamment naturel et l'absence de clicks le rend plus fluide par rapport à un menu où il est nécessaire de cliquer. Pour cette raison, nous avions pensé à une version sans mode (ou transitoire) où le menu disparaîtrait pendant l'utilisation de l'opération de navigation puis réapparaîtrait à la fin de l'opération. Dans cette version, pour éviter les aller-retour entre le lieu d'utilisation de l'opération et l'emplacement du menu, mais également pour exploiter la fluidité du geste nécessaire à son activation, le menu aurait pu réapparaître dans le voisinage du lieu de relâchement de l'opération (endroit où le stylet quitte la surface) pour pouvoir rapidement activer une autre opération (peut-être la même que celle qui vient juste d'être utilisée si plusieurs mouvements sont nécessaires).

Lors de l'essai de cette variante du menu à barres par les personnes du laboratoire, nous avons observé que la disparition puis la réapparition du menu à un emplacement différent était fort déconcertante et ne permettait pas du tout le mouvement fluide et le passage rapide à une autre opération comme escompté.

5.3 Appel du widget

Un des prototypes de widgets a été intégré au logiciel de dessin collaboratif SketSha. Avec SketSha, plusieurs personnes utilisant des ordinateurs différents peuvent partager un même document. Le point de vue est identique pour chacun des participants. Les opérations de dessin ou de navigation peuvent être réalisées par n'importe

quel utilisateur et les autres verront le changement immédiatement.

Lors d'un travail pratique, des étudiants en architecture ont utilisé SketSha pour élaborer un projet architectural. Les étudiants devaient travailler en quatre groupes. Chaque groupe est constitué d'étudiants belges et d'étudiants français. Les étudiants belges faisaient usage du Bureau Virtuel installé au laboratoire du Lucid Group et les étudiants français utilisaient un second Bureau Virtuel installé en France, à l'Ecole Supérieure d'Architecture de Nancy. Ce travail a duré trois mois. Pendant les deux premiers mois, la navigation fonctionnait suivant le principe d'un menu fixe et le point de référence pour le zoom et la rotation n'était pas manipulables. Le dernier mois, un des prototypes de widgets a été utilisé à la place du menu fixe. La position du widget définissait le point de référence. Les opérations étaient activées et désactivées par trois boutons. Un petit bouton supplémentaire permettait de ramener la caméra à sa position de départ, au centre de la scène, alignée sur les axes principaux et avec le facteur de zoom ramené à un. Il était également permis d'appeler le widget en appuyant le stylet et en attendant un court instant sans bouger.

Lors de cette expérience, nous avons constaté que ces possibilités étaient utilisées et appréciées des étudiants. Nous avons aussi pu remarquer que le temps nécessaire pour réaliser l'appel du widget (300 millisecondes) était trop court. Il est arrivé en effet qu'en commençant un trait sur le papier virtuel, le widget soit appelé alors que l'utilisateur souhaitait simplement dessiner.

5.4 Modifications de SketSha

En vue de l'évaluation des différents widgets développés, une version modifiée de SketSha a été créée. Le but poursuivi est de pouvoir soumettre à des utilisateurs une scène contenant quelques figures et de leur demander de déplacer la caméra afin d'atteindre un point de vue prédéterminé. Celui-ci est affiché, fixe, par dessus le reste de la scène. Il faut donc que les traits des figures de la scène et ceux des figures fixes correspondent. Les modifications apportées enregistrent les mouvements et les contacts avec la surface du stylet. Les enregistrements ainsi obtenus pour chaque utilisateur auraient permis de connaître les détails d'utilisation du widget : le temps nécessaire à atteindre l'objectif, les opérations utilisées (y compris plusieurs fois la même de suite) ou les distances parcourues (en pixels). Ces données auraient pu être complétées par un questionnaire pour connaître le sentiment de l'utilisateur vis-à-vis du widget.

Chapitre 6

Moteur dataflow

La réalisation des différents modèles de widgets pour la navigation dans une scène 2D et dans une scène 3D des chapitres précédents a été rendu plus compliquée que nécessaire par l'intégration continue des widgets dans une application en cours de développement dans le laboratoire de recherche. Dans ce chapitre, une bibliothèque, appelée AEDF, est développée pour le langage de programmation Scheme. Cette bibliothèque s'utilise pratiquement comme un langage de programmation à part entière et permet d'écrire de façon succincte des programmes graphiques interactifs. En particulier, il est montré qu'il est possible d'écrire avec cette bibliothèque une application très simple de dessin pouvant servir de support à l'évaluation de widgets de navigation, également écrits avec cette bibliothèque. Le programme final exprime de manière concise le fonctionnement de l'application.

L'intérêt premier de cette bibliothèque, et éventuellement d'un langage basé dessus, est la possibilité pour le programmeur d'exprimer facilement un programme interactif. Le programme peut être gardé en l'état ou réécrit s'il s'avère que la bibliothèque ne possède pas les performances requises par l'application. Néanmoins, nous pensons qu'en compilant le programme sous une forme plus compacte, le problème des performances ne devrait pas subsister.

AEDF est un moteur d'évaluation dataflow. Il est un proche cousin de FrTime[CK06], une extension pour DrScheme[FCF$^+$02] elle-même inspirée de la programmation fonctionnelle réactive (FRP)[HCNP03]. Le principe de l'évaluation dataflow est bien connu : c'est celui employé dans les logiciels de feuilles de calcul (*spreadsheets* en anglais). Le logiciel gère un ensemble de variables. Lorsque l'une des variables voit sa valeur changer, toutes les variables qui en dépendaient sont recalculées automatiquement.

Le but principal de FrTime est de garder les symboles existants de Scheme pour pouvoir graduellement modifier un programme écrit en Scheme en un programme interactif. Tout programme Scheme est un programme FrTime équivalent. Pour cela, les procédures existantes sont modifiées pour accepter aussi bien les valeurs Scheme traditionnelles que les valeurs manipulées par FrTime. Ce choix permet effectivement de garder la "syntaxe" des programmes Scheme existants au prix d'une perte de performance.

Un programme écrit avec AEDF est réactif. L'exécution du programme dépend des évènements reçus en entrée. Typiquement, ces évènements proviennent de la frappe au clavier ou de l'utilisation de la souris. Non seulement l'évolution du programme dépend des entrées, mais il peut également dépendre de l'écoulement du temps, par exemple pour animer des objets à l'écran ou faire la différence entre deux simples clicks consécutifs et un double-click.

Fondamentalement, un programme AEDF manipule des objets appelés *signaux*. Les signaux sont des valeurs qui évoluent avec le temps. La bibliothèque fournit toute une série de signaux dits de base. Ces signaux de base représentent principalement le temps ainsi que les évènements utilisateurs comme la frappe au clavier ou les mouvements de la souris. Par exemple, un des signaux de base représente le temps écoulé depuis le lancement du programme. En plus de ces signaux, AEDF fournit des opérateurs que l'on peut appliquer à des signaux pour en construire de nouveaux. Ecrire un programme AEDF revient à former des expressions en utilisant les signaux de base et les opérateurs pour définir d'autres signaux. Parmi les signaux que le programmeur peut former, se trouvent des signaux de sortie permettant d'observer l'évolution du programme. Les signaux de sorties que l'on peut construire comprennent des objets graphiques tels des rectangles ou des labels (un label est un texte entouré d'un bord rectangulaire). Tout au long de l'exécution d'un programme AEDF, la bibliothèque garantit de maintenir les relations définies entre signaux par le programmeur.

6.1 Exemple de programme

Pour mieux percevoir comment on peut écrire une application interactive avec AEDF et pour avoir une impression générale de la bibliothèque, nous présentons un court exemple (fig. 6.1) avant de décrire son fonctionnement et son implémentation en détail. Deux captures d'écran sont montrées dans la figure 6.2.

Dans cet exemple, nous affichons à l'écran un label. Ce label est positionné au même emplacement que le curseur de la souris. Le texte du label affiche les coordonnées de ce même curseur.

```
(use aedf)

(label
  'text (s-string "~s,~s" mouse-x mouse-y)
  'position mouse-position)

(aedf:run)
```

FIG. 6.1 – *Exemple de programme AEDF. Un label est positionné au même emplacement que le curseur.*

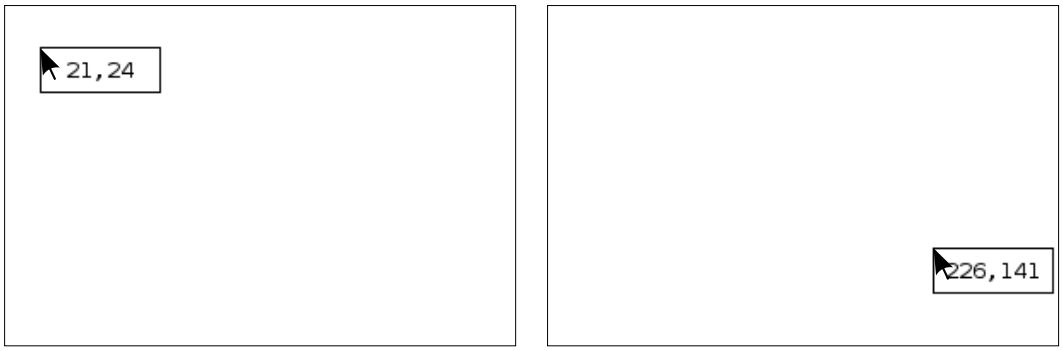


FIG. 6.2 – *Cette figure présente deux captures d'écran de l'exécution du programme de la figure 6.1. Le label est placé au même endroit que le curseur et en affiche les coordonnées.*

La première ligne permet d'importer la bibliothèque AEDF rendant les constructions qu'elle définit disponibles pour écrire un programme AEDF. Les trois lignes qui suivent forment le programme AEDF proprement dit. La dernière ligne démarre l'exécution du programme.

Le programme fait usage des opérateurs `label` et `s-string` et des signaux de base `mouse-x` et `mouse-y`.

La procédure `label` crée un nouveau signal de type 'label'. Un label est formé d'attributs construits à partir d'autres signaux donnés en arguments. Les arguments ne sont pas donnés en fonction de leur position mais nommément, en les faisant précéder d'un symbole identifiant leur rôle. AEDF permet de spécifier uniquement les attributs qui nous importent, laissant aux autres leur valeur par défaut. Ici, nous spécifions la position (du coin supérieur gauche) du label ainsi que son texte. La position du label est `mouse-position`. `mouse-position` est un signal fourni par la bibliothèque qui vaut en permanence la position du curseur de la souris. Nous

écrivons donc que la position du label est en permanence la position du curseur. Cela est suffisant pour qu'à l'exécution, peu importe où se trouve le curseur, le label se trouve à la même position que lui. Le signal `mouse-position` est en fait une paire de deux signaux de base : la position selon l'abscisse et la position selon l'ordonnée du curseur. Ces deux signaux sont `mouse-x` et `mouse-y`. Dans l'exemple, ils sont aussi utilisés pour écrire le signal qui constitue le texte du label grâce à la procédure `s-string`. Cette procédure prend au moins un argument : une chaîne de caractère. Cette chaîne de caractère peut indiquer qu'il y a d'autres arguments à afficher avec le caractère '~'. Le caractère qui suit le tilde permet d'indiquer la façon dont l'argument est affiché. `~s` permet d'afficher une valeur Scheme de manière lisible pour un être humain.

L'évolution d'un programme AEDF se fait par étapes. A chaque étape, le programme passe d'un état à un autre. Le nouvel état résulte du traitement d'un évènement d'entrée et dépend de l'état courant. Pour que l'évolution soit visible, il faut que le programme possède des sorties. La sortie la plus courante est une animation à l'écran formée par l'affichage d'image en succession rapide. Autrement dit, les images affichées peuvent être considérées comme étant un signal particulier qui dépend de tous les signaux représentant des objets graphiques tels les rectangles et les labels. Une autre sortie est la console. Elle contient une suite croissante de caractères. D'autres sorties sont envisageables comme des sons ou des messages envoyés sur le réseau.

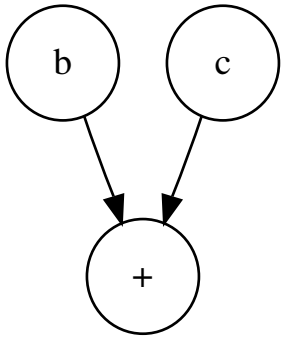
6.2 Signaux

Dans un langage impératif, l'abstraction fondamentale du langage est la fonction (ou la procédure) et dans un langage orienté-objet, l'abstraction mise en avant par le langage est la classe. Dans le modèle dataflow adopté ici, la brique de base est le signal. Un signal est une valeur qui évolue avec le temps. Un programme dataflow est une collection de signaux organisés selon un graphe dirigé. A cause de cette organisation, les signaux sont aussi appelés noeuds lorsque l'on réfère à leur rôle dans le graphe. Les arcs entre noeuds représentent des dépendances. Un arc démarre depuis un noeud parent et arrive à un noeud enfant et l'on dit que la valeur de l'enfant à un moment donné dépend des valeurs de ses parents à ce même moment.

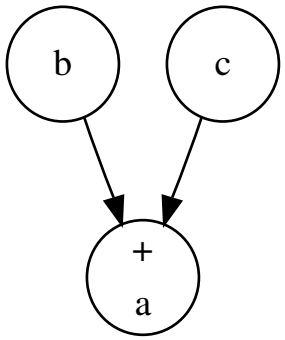
Les cycles dans le graphe ont une signification particulière et le graphe peut être considéré comme acyclique. De ce fait, il existe trois sortes de noeuds suivant la position prise dans le graphe. Le noeud de base ne possède pas de parents (aucun arc ne se termine à ce noeud). Le noeud feuille ne possède pas d'enfants (aucun arc ne démarre de ce noeud). Enfin, le noeud intérieur est le noeud qui n'est ni un noeud de base, ni une feuille (un ou plusieurs arc(s) arrive(nt) à ce noeud et un ou

plusieurs arc(s) le quitte(nt)).

Le graphe correspond à des opérations sur des signaux. Par exemple, l'addition de deux signaux `b` et `c` correspond à un troisième signal. Un tel graphe est similaire aux arbres syntaxiques abstraits rencontrés en compilation de langage de programmation. Visuellement, le graphe est représenté à l'envers : les deux noeuds dont dépend l'addition sont situés en haut et l'opérateur d'addition est situé en bas.



`b` et `c` sont les noms portés par les noeuds dont dépend le noeud `+`. Le noeud `+` est un noeud feuille : aucun autre noeud ne dépend de lui. Si l'on désire montrer dans le dessin que ce dernier est nommé, le nom est placé en dessous de l'opération.

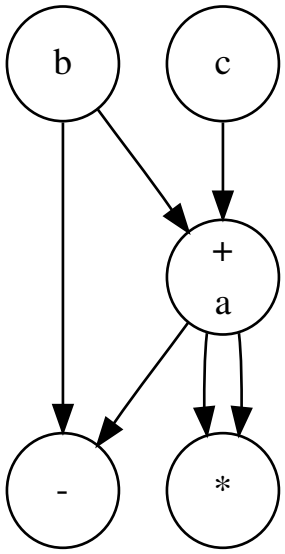


En Scheme, l'addition du nombre `b` et du nombre `c` est écrite `(+ b c)`. Toute expression commence par une parenthèse ouvrante et termine par une parenthèse fermante. Le premier symbole (ou de manière générale, la première sous-expression) d'une expression doit être une procédure, comme `+`, ou une forme spéciale, comme `define` ou `if`. De façon similaire, le graphe ci-dessus est écrit avec la bibliothèque

dataflow `(s+ b c)`. Pour nommer le signal résultant, la procédure `node` [1] est utilisée : `(node a (s+ b c))`. Cette expression peut être vue conceptuellement par le programmeur Scheme comme l'équivalent de `(define a (+ b c))`.

Nommer un signal permet de l'utiliser dans plusieurs sous-expressions différentes. Contrairement à la forme spéciale `define`, la macro `node` n'introduit pas une nouvelle portée lexicale. Il est donc possible de nommer un signal à l'intérieur d'une expression et de l'utiliser dans une autre expression. Effectivement, chaque symbole introduit par `node` est disponible globalement. Dans l'état actuel de la bibliothèque, aucun système de module n'a été introduit.

```
(s- b (node a (s+ b c)))
(s* a a)
```



6.2.1 Exécution d'un programme dataflow

L'évolution d'un programme dataflow repose sur la notion de coupe (ou *slice* en anglais). Dans AEDF, le temps s'écoule de manière discrète. Chaque point dans le temps donne lieu à une ou plusieurs coupes. L'état observable du programme, c'est-à-dire les valeurs prises par l'ensemble des signaux qui le définissent, est figé à chaque coupe. En particulier, les coupes pour effectuer le rendu capturent graphiquement l'état du programme à ce moment. Ces coupes correspondent aux signaux

[1] `node` est en fait implémentée comme une macro.

de sorties et peuvent exploiter les signaux dont ils dépendent. Il y a également des coupes qui correspondent aux signaux de base.

Les valeurs prises par les signaux sont calculées suivant les opérations utilisées pour créer le graphe. A chaque coupe d'entrée, certains des signaux au sommet du graphe reçoivent une nouvelle valeur et le reste du graphe est mis à jour. Les valeurs résultantes forment l'état du programme pour la coupe.

6.2.2 Boucles et délais

Dans la section sur les signaux, il a été vu qu'il est possible de nommer un signal pour pouvoir l'utiliser dans une expression avec la macro `node`. Une utilisation courante de cette possibilité est la définition d'un signal en fonction de lui-même. Néanmoins, une expression comme `(node a (s+ a b))` n'a pas de sens dans le modèle dataflow adopté par AEDF puisqu'il ne s'agit pas d'une affectation mais bien d'un invariant qui doit resté vérifié à tout moment de l'exécution du programme. De ce fait, la seule expression récursive qui pourrait être vérifiée est `(node a a)`.

Dans un langage synchrone, comme Esterel[Ber00] par exemple, cette limitation existe également et est appelée *boucle instantanée*. Mais cette limitation est réduite en permettant à un signal de dépendre d'une valeur passée de lui-même, c'est-à-dire en créant une boucle étalée dans le temps. Ce principe est repris ici via l'opérateur `s-delay`. A chaque coupe, un noeud `(s-delay a)` a pour valeur celle prise par le noeud `a` à la coupe précédente. Bien entendu, il est possible d'utiliser l'opérateur `s-delay` sur un signal résultant lui-même de l'application de ce même opérateur. Ainsi, on peut construire un signal possédant la valeur d'un noeud plusieurs coupes auparavant de la coupe courante.

L'interdiction d'avoir des boucles instantanées dans le graphe est la raison pour laquelle il est dit plus haut que le graphe est acyclique. Les cycles ne font pas véritablement intervenir deux fois le même signal : le noeud `(s-delay a)` ne dépend pas de la valeur de `a` au moment présent. Il peut être vu comme un signal de base (qui ne possède pas de parents) mais néanmoins associé au signal `a`. De manière générale, tout signal possède un tel noeud associé.

Ceci dit, quelle valeur prend un noeud signal à la première coupe du programme ? Ou plus généralement, quelle valeur donner au passé d'un signal (puisqu'il est possible d'empiler les applications de l'opérateur `s-delay` pour remonter dans le passé de plusieurs coupes) ? La réponse est de considérer les signaux comme constants jusqu'au démarrage du programme. Si le signal est constant, on sait que sa valeur dans le passé est la même qu'au moment présent. De cette manière, la valeur d'un signal délai est la même que le signal auquel il est associé. Dans le cas où un si-

gnal dépend de son passé, la valeur initiale doit être donnée explicitement avec la procédure `initial-value`.

6.2.3 Evènements et comportements

Un signal est une valeur qui évolue avec le temps. Ou plutôt, dans le modèle discret d'AEDF, un signal possède une valeur qui évolue à chaque coupe. En plus de cette valeur, un signal est caractérisé par son absence ou sa présence pendant une coupe.

La notion de présence (et d'absence) permet de modéliser dans un programme AEDF à la fois des signaux représentant des évènements discrets et des signaux conceptuellement continus, c'est-à-dire définis à tous points dans le temps.

Traditionnellement, ces deux formes de signaux reçoivent en programmation fonctionnelle réactive les noms d'évènements et de comportements. Un comportement est un signal qui est présent en permanence, c'est-à-dire à chaque coupe. Par exemple, le signal représentant le temps écoulé est un comportement. Un évènement (ce terme ne recouvre pas la même chose que les évènements d'entrée comme les mouvements de la souris) est un signal qui n'est présent que lorsque l'évènement qu'il représente a effectivement lieu. Par exemple, l'évènement touche-enfoncée n'est présent que dans la coupe associée au traitement de l'enfoncement d'une touche (et il sera absent notamment à la coupe de traitement du changement de position de la souris). Notez bien qu'un évènement est un signal. Chaque fois que le signal est présent, on dit qu'il y a une occurrence de l'évènement.

Dans AEDF, cette distinction, bien qu'utile pour s'exprimer, n'est que conceptuelle. Même si la bibliothèque était implémentée dans un langage fortement typé, les deux sortes de signaux recevraient le même type. Aussi, il n'y a pas de lien particulier entre évènements et effets de bord. On peut d'ailleurs définir un comportement comme un évènement possédant une occurrence à chaque coupe ou, dans le sens contraire, définir un évènememement comme un comportement dont le domaine des valeurs qu'il peut prendre est augmenté de la valeur `'undefined` ou d'un autre symbole particulier.

Considérons à nouveau l'expression de la section précédente : `(node a (s+ b c))`. Elle dénote un signal, nommé `a`, qui est défini comme possédant pour valeur la somme des valeurs des signaux `b` et `c`. Cette définition est maintenue tout au long de l'exécution du programme. Peu importent les valeurs que prennent `b` et `c` à un moment donné, `a` vaudra toujours la somme de `b` et `c`. Si à un moment t_1, `b` et `c` valent respectivement 2 et 3, alors `a` vaudra 5 au moment t_1. Si au moment t_2, `b` et `c` valent 12 et 74, alors `a` vaudra 86 au moment t_2.

6.3 Boucle principale

Exécuter un programme AEDF se fait en appelant la procédure `aedf:run`. Après un temps d'initialisation, `aedf:run` entre dans une boucle et n'en sort que lorsque l'utilisateur décide de terminer le programme (généralement en appuyant sur la touche Echap). Cette boucle est caractéristique des programmes interactifs et est appelée simplement boucle principale. En anglais, il s'agit de *main loop* ; dans un jeu vidéo, on parlera généralement de *game loop*.

6.3.1 Coupes

C'est cette boucle qui fait évoluer le programme par étape. Chaque étape est appelée 'coupe' et correspond soit à la modification d'un signal d'entrée (ou encore appelé signal de base) soit à l'utilisation des valeurs des signaux de sorties (typiquement pour réaliser le rendu). Chaque coupe affectant un signal d'entrée effectue une mise à jour des signaux qui en dépendent. Si nous considérons à nouveau l'expression `(node a (s+ b c))` en imaginant que `b` est un signal de base, la coupe qui modifie `b` réalisera une mise à jour de `a`. Lorsque une coupe de sortie utilisera les valeurs de `a`, `b` et `c`, ces valeurs vérifieront bien l'invariant $a = b + c$.

Les coupes sont variées en fonction des évènements d'entrée et des types de sorties gérés par la boucle. Typiquement, un programme réactif doit réagir au temps qui passe, aux entrées clavier et souris et réaliser un rendu à l'écran. Il faut donc des signaux de base pour représenter le temps qui passe et l'utilisation du clavier et de la souris, ainsi que des coupes de sorties pour le rendu. On peut aussi imaginer une boucle qui ne possède que des coupes d'entrée et de sortie associées à des messages émis sur le réseau. Suivant la complexité du rendu, plusieurs coupes de sortie peuvent être nécessaires, par exemple une coupe pour le rendu d'une scène 3D et une autre pour le rendu d'une interface graphique affichée en transparence par dessus l'image de la scène.

6.3.2 Découpage de la boucle

Comme mentionné plus haut, une boucle peut incorporer différents types de coupes suivant l'application à réaliser (graphique, réseau, son, ...), c'est-à-dire suivant le type des signaux d'entrée à traiter et suivant les types de signaux de sortie désirés. Mais il est également à noter que les coupes ne sont pas systématiquement réalisées à chaque itération de la boucle. Si aucun évènement clavier ne survient pendant une itération, aucune coupe associée au traitement des évènements clavier ne sera exécutée. De même, s'il y a plusieurs évènements clavier ou souris en attente au

moment où la boucle traite les entrées clavier et souris, plusieurs coupes seront exécutées (une coupe par évènement). De ce fait, les évènements en entrée sont traités séparément et ne sont jamais présents en même temps. Une autre modélisation de la boucle aurait été de collecter les changements d'états des signaux associés aux entrées puis de faire la mise à jour des signaux qui en dépendent. Ce choix n'a pas été retenu pour la raison suivante : il est nécessaire de pouvoir traiter les évènements en entrée séparément car l'ordre d'apparition des évènements ainsi que les occurrences successives d'un même évènement peuvent avoir leur importance suivant ce que l'on désire implémenter.

Pour bien se rendre compte de cette importance, considérons deux situations. La première consiste en l'affichage à l'écran d'un rectangle. Normalement, le rectangle est fixe. Mais lorsque l'utilisateur appuie sur la touche 'espace', le rectangle doit se mettre en mouvement et se déplacer vers la droite. Le rectangle cesse de se déplacer une fois que l'utilisateur n'appuie plus sur la touche espace. Pour implémenter cela, il suffit d'avoir un signal booléen de type comportement (présent à chaque coupe) qui vaut soit *vrai* si la touche est enfoncée, soit *faux* dans le cas contraire. Ensuite, il faut associer comme position du rectangle deux possibilités selon la valeur du signal booléen. S'il est faux, le rectangle conserve sa position ; s'il est vrai, le rectangle prend une position dont la valeur est la position au rendu précédent additionnée d'un facteur de translation. Ainsi, dans ce second cas, à chaque rendu le rectangle se retrouve un peu plus loin de sa position initiale.

La seconde situation est analogue mais au lieu d'avoir un rectangle qui se déplace tant que la touche espace est enfoncée, le rectangle doit se déplacer 'par bonds' chaque fois que la touche passe de l'état 'relevé' à l'état 'enfoncé' (l'expression 'la touche est enfoncée' peut être comprise de deux façons : la touche n'est pas dans l'état 'relevé' et est maintenue dans cette position ou alors la touche passe de l'état 'haut' à l'état 'bas' ; ici, il s'agit de la seconde). Appuyer à deux reprises doit donc avoir pour effet de faire effectuer deux bonds au rectangle.

La deuxième situation ne pose pas de problème et peut être gérée avec une seule coupe pour traiter tous les évènements disponibles. Par contre, la première situation ne donnerait pas le résultat désiré si une seule coupe était utilisée. Si une seule coupe traite tous les évènements disponibles, alors il se peut que les évènements touche-espace-enfoncée et touche-espace-relevée soient tous deux présents. La seconde situation ne se préoccupe que du signal associé à touche-espace-enfoncée et peut réagir en modifiant l'emplacement du rectangle. La première situation utilise un signal booléen qui passe d'un état à un autre suivant la présence de touche-espace-enfoncée ou touche-espace-relevée. Mais quelle valeur donner à ce signal lorsque ces deux derniers sont présents dans la même coupe ? La réponse dépend de l'ordre dans lequel ces évènements d'entrée ont été générés. Or cette information n'est pas disponible : tous les évènements dans une coupe sont simultanés. Comme

l'ordre d'apparition des évènements d'entrée est important dans certaines situations (comme la première décrite), il est nécessaire d'utiliser une coupe séparée pour chacun d'entre-eux.

Pour finir, imaginons une situation où l'on désire avoir un signal qui compte les occurrences d'un évènement d'entrée. Si cet évènement est présent plusieurs fois dans la file d'attente alors qu'il n'y a qu'une coupe, il ne sera pas possible de distinguer les différentes occurrences : au niveau du graphe, l'évènement est présent ou non mais il ne peut être présent de multiples fois. Suivant le type d'application à réaliser, il est donc possible qu'un autre type de boucle soit plus adapté.

6.3.3 Mise à jour synchrone

A l'intérieur d'une coupe, tout signal est soit présent soit absent. Même si le processus de mise à jour travaille sur un signal à la fois, conceptuellement l'exécution de la coupe est atomique. Il n'est pas possible d'observer grâce à un signal de sortie l'évolution de la coupe mais bien uniquement l'état qui en résulte. La mise à jour est menée de manière à assurer que les définitions des signaux soient respectées. Un programme AEDF est un ensemble de signaux dont les relations entre eux sont autant d'invariants.

Une coupe est le plus petit découpage de temps possible pour un programme AEDF. Cela signifie que plusieurs évènements présents (ou absents) dans une coupe le sont simultanément. Il n'y a pas d'ordre dans les occurrences de ces évènements (c'est pour cela que s'il est nécessaire d'observer un ordre entre deux évènements d'entrée, il faut deux coupes).

6.3.4 Réaction à l'absence d'un évènement

Contrairement à d'autres implémentations de programmation réactive, il est possible de réagir à l'absence d'un évènement. Pour décider qu'un évènement est absent, il suffit de connaître quels éléments du programme sont susceptibles de le générer. En exécutant ces éléments avant ceux qui dépendent de l'absence ou de la présence de l'évènement, on peut être certain que si l'évènement n'a pas été généré, il ne le sera plus pendant la coupe courante. L'implémentation d'AEDF reposant sur un graphe de dépendances respecte ces deux conditions : les noeuds du graphes sont mis à jour dans l'ordre de leur rang. Lors du calcul de la valeur courante d'un noeud qui dépend d'un évènement, la présence ou l'absence de ce noeud a déjà été décidée (comme ce noeud est parent du noeud à calculer, il possède un rang inférieur et a donc déjà été mis à jour pour la coupe courante).

6.4 Usage

La bibliothèque AEDF est écrite comme une extension pour Chicken, une implémentation du langage Scheme[2]. Chicken fournit un compilateur utilisant C comme langage intermédiaire ainsi qu'un interpréteur. Chicken possède plusieurs caractéristiques qui le rendent intéressant pour réaliser une telle bibliothèque.

Scheme, comme tout dialecte de LISP, possède une syntaxe simple et constante, les s-expressions. Cette syntaxe reflète bien la structure en graphe des expressions d'AEDF et permet de rendre l'utilisation de la bibliothèque AEDF proche de celle d'un langage à part entière. Scheme permet par exemple de nommer une procédure `s+` pour créer un signal qui additionne les valeurs des deux signaux dont il dépend. S'il s'avère que les procédures fournies par R5RS comme `+`, `sin` ou `abs` ne sont pas nécessaires pour écrire un programme AEDF, rien n'empêche la bibliothèque de réutiliser ces symboles à la place de `s+`, `s-sin` ou `s-abs`. Pour compléter l'illusion, un simple script nommé `aedf` permet d'abstraire la première et la dernière ligne de l'exemple de la figure 6.1. Ensuite on peut écrire un programme AEDF et oublier qu'il repose sur Chicken. Etant donné un fichier nommé `example-01.aedf` reprenant les trois lignes centrales de l'exemple, le script `aedf` appelé sur ce fichier est équivalent à :

```
csi -R aedf example-01.aedf -e "(aedf:run)"
```

`csi` est le programme *chicken scheme interpreter*. L'option `-R` permet de charger une extension (dans ce cas, il s'agit de la bibliothèque AEDF) et l'option `-e` permet de demander l'exécution d'une expression. Ici l'expression permet d'exécuter le graphe dataflow en appelant `aedf:run`.

Chicken est implémenté en utilisant le langage C comme langage intermédiaire et propose plusieurs extensions facilitant l'écriture de programmes ou de bibliothèques mélangeant C et Scheme. En particulier, il est relativement aisé d'utiliser une bibliothèque C depuis Chicken. Cette possibilité est particulièrement intéressante puisque de nombreuses bibliothèques sont originalement disponibles pour C. C'est le cas par exemple des bibliothèques OpenGL et Xlib utilisées pour implémenter AEDF. Bien sûr cet élément en faveur de Chicken peut être transposé à une implémentation utilisant, par exemple, la JVM si l'on préfère utiliser les nombreuses bibliothèques disponibles pour Java.

L'existence d'un interpréteur permet d'écrire et d'exécuter rapidement des fragments de programme puisqu'il n'y a pas à attendre la fin de la compilation. Elle permet

[2]Chicken supporte essentiellement R5RS[KCe+98].

également de tester interactivement la bibliothèque en cours d'écriture ou des programmes AEDF utilisant cette bibliothèque. Potentiellement, l'interpréteur pourrait être utilisé pour écrire interactivement un programme AEDF et le modifier pendant son exécution.

6.5 Relation avec la programmation multi-threads

6.5.1 FairThreads

Le modèle de mise à jour synchrone des signaux est inspiré des FairThreads. Le modèle des FairThreads a été développé pour fournir un mécanisme de concurrence simple à utiliser. Il a été inspiré par l'approche réactive et il fournit des primitives pour écrire des programmes concurrents. Les FairThreads sont des threads non-préemptés. Cela signifie qu'un thread qui a le processeur ne le quittera que de sa propre initiative. S'il y a plusieurs threads qui se partagent le processeur, ils doivent être écrits de façon à ce qu'ils coopèrent. Les FairThreads communiquent par le biais d'évènements. Un évènement est une structure de données contenant une valeur pour l'instant courant et un drapeau indiquant si l'évènement est présent. Chaque thread est en réalité une machine à états. L'évolution du programme se fait de façon discrète (le temps est découpé en instants) en faisant progresser chaque machine à états. Un thread peut attendre qu'un évènement soit présent, exécuter du code écrit en C, émettre un évènement ou attendre l'instant suivant.

Les threads sont exécutés tour à tour. Un thread relâche le processeur lorsqu'il attend un évènement ou lorsqu'il attend l'instant suivant. Lorsqu'un thread attend qu'un évènement survienne, il est donc arrêté pour l'instant courant. Mais si un thread exécuté par la suite, toujours dans le même instant, émet l'évènement, le thread en attente est réexécuté, continuant à partir de l'état où il se trouvait. C'est de cette manière que le caractère synchrone est atteint : il est possible d'émettre un évènement à condition qu'un autre évènement soit présent. Les deux évènements seront alors tous deux présents au cours du même instant.

6.5.2 AEDF et FairThreads

Pour voir le rapport entre le modèle des signaux d'AEDF et celui de la programmation concurrente des FairThreads, intéressons-nous aux signaux que l'on appelle évènements (les signaux qui ne sont présents que pendant certaines coupes).

Dans AEDF, on peut voir un évènement construit à partir d'un autre évènement

comme un thread très simple. Par exemple, étant donné un évènement `nbrs` qui fournit des nombres entiers, on peut construire un second évènement `odds` qui est présent uniquement lorsque le nombre émis par le premier est impair.

```
(node odds (s-odd nbrs))
```

Cette association entre deux objets représentant des évènements peut être obtenue avec les FairThreads, de façon simplifiée, comme ceci :

```
loop
  wait nbrs;
  if nbrs' current value is odd
  then emit odds;
end-loop
```

Il s'agit d'un thread qui attend l'occurrence de l'évènement `nbrs` puis qui émet l'évènement `odds`.

Autrement dit, un sous-graphe établissant les définitions d'évènements à partir d'autres évènements peut être vu comme l'équivalent d'un thread. Un opérateur comme `odd` construit un noeud à partir d'un autre. Un thread définit un ou plusieurs évènement(s) à partir d'un ou plusieurs évènement(s). Effectivement, la modularité obtenue (ou cherchée) par la programmation concurrente est obtenue en composant plusieurs sous-graphes en un seul programme dataflow. Chaque sous-graphe peut être écrit indépendamment des autres et ils s'exécuteront parallèlement.

Parce que le lien entre le code de mise à jour d'un évènement et l'évènement est plus étroit dans le cas d'AEDF, une fois qu'un évènement est défini, son évolution sera toujours la même indépendamment des sous-graphes qui peuvent être ajoutés au programme complet. Dans FairThreads, un évènement est véritablement une structure de données partagée. Il n'y a pas de restriction sur les threads qui peuvent le lire ou l'écrire. De plus, toujours dans le cas d'AEDF, les dépendances entre "threads" et évènements sont connues. C'est ce qui permet de savoir au cours d'une coupe qu'un évènement est absent. Dans FairThread, l'absence d'un évènement pendant un instant ne peut être connue d'un thread qu'au cours de l'instant suivant. Cette limitation est nécessaire dans le cas de FairThread car l'existence des threads pendant l'exécution est complètement dynamique : ils peuvent naître et mourir pendant l'exécution sans restriction sur leur nombre.

6.6 Reconfiguration dynamique du graphe

La bibliothèque AEDF permet de facilement écrire un programme qui fait évoluer des valeurs en fonction du temps et d'évènements d'entrée. Une limitation importante à l'heure actuelle de la bibliothèque est qu'il n'est pas possible de modifier la structure du graphe pendant l'exécution. La définition du graphe se fait par l'ajout progressif de noeuds. Une fois le graphe défini, le programme est exécuté avec la procédure `aedf:run`. Lorsque cette procédure est appelée, le graphe est utilisé pour faire évoluer l'état du programme et il n'est plus possible d'ajouter de nouveaux noeuds. Il n'est pas possible non plus d'en retirer ou d'en déplacer.

Concrètement, il existe trois situations où cette limitation se fait sentir. Cependant, dans les trois cas, il est possible de contourner le problème. Le premier cas est l'ajout dynamique (à l'exécution) de nouveaux noeuds de sortie. La seconde situation est l'évaluation d'un noeud de type `if`. Le troisième cas est semblable au premier mais avec une simplification : le nombre de noeuds de sorties pouvant être ajoutés est connu statiquement (au moment de construire le graphe, avant de lancer l'exécution).

Considérons d'abord la troisième situation. Imaginons que l'on veuille permettre à l'utilisateur de placer des rectangles à l'écran, à la position voulue. Initialement, il n'y aucun rectangle affiché. Un click ajoute un rectangle à la position du curseur. Un click où se trouve un rectangle supprime le rectangle. Pour satisfaire la simplification par rapport à la première situation, nous limitons le nombre de rectangles, par exemple à 5. Pour implémenter ce programme, il est nécessaire d'avoir un noeud de sortie de type rectangle pour chaque rectangle que l'utilisateur peut placer. Initialement, les rectangles ne sont pas visibles et lorsque l'utilisateur place un rectangle, il suffit d'en rendre un visible. Plutôt que d'écrire un programme qui ajoute des noeuds de sortie de type rectangle à l'exécution, provision est faite au moment de construire le graphe pour le nombre maximum de rectangles possible. Ajouter ou retirer un rectangle n'est pas implémenté en ajoutant ou retirant un noeud au graphe mais en le rendant visible ou non. Le caractère visible ou non d'un rectangle est un signal booléen.

Si le nombre de rectangles que l'utilisateur peut placer à l'écran n'est pas limité, la situation ne peut pas être implémentée avec la bibliothèque dans l'état. Il est nécessaire d'avoir un noeud de sortie pour chaque rectangle placé or il n'est pas possible de prévoir tous ces noeuds au moment de la construction du graphe. Il faudrait donc étendre les possibilités de la bibliothèque. Une extension possible est la modification dynamique du graphe. Mais une autre possibilité qui ne nécessite pas d'ajout important à la bibliothèque est envisageable. A la place d'ajouter une capacité générique (modifier le graphe), on peut ajouter un nouveau type de noeud

de sortie. Ce type de noeud possède pour parents les signaux de position et de click de la souris et maintient l'état de l'ensemble des rectangles à afficher. En fonction des évènements, l'état est modifié en ajoutant ou en retirant des rectangles. Ce noeud est en outre responsable de l'affichage des rectangles. Cette solution n'est pas très élégante car elle demande de gérer une nouvelle structure de données ainsi que l'écriture de code pour le rendu. C'est pourquoi rendre le graphe dynamique est nécessaire pour faire d'AEDF une bibliothèque souple à l'emploi. Puisqu'un objet graphique est représenté par un noeud, il est plus naturel d'ajouter des objets graphiques par l'ajout de noeuds que par l'implémentation d'un nouveau type de noeud.

La seconde situation nécessitant à priori un graphe dynamique est l'évaluation de noeud de type `if`. Normalement, en programmation, une seule des deux branches de code gardées par le `if` est exécutée : la première dans le cas où la condition du `if` est évaluée à vrai, la seconde dans le cas contraire. Dans FrTime, cette propriété est conservée en modifiant le graphe à l'exécution. En effet, le modèle d'évaluation des valeurs des noeuds propage normalement les changements à partir du haut du graphe en direction des enfants, en les mettant à jour pour la coupe courante. Dans ce modèle, une expression comme `(s-if condition branche1 branche2)` représente un noeud dont les parents sont `branche1` et `branche2`, exactement de la même manière que le noeud `(s+ a b)` dépend des noeuds `a` et `b`. Si l'on veut évaluer uniquement la `branche1` ou `la branche2` en fonction de la valeur du noeud condition, cela demande qu'une des deux branches appartienne au graphe et pas l'autre (pour qu'une seule des deux soit mise à jour pendant une coupe). Or l'évaluation de la condition n'a lieu que pendant la coupe, donc l'ajout ou la suppression se fait également pendant la coupe.

Dans AEDF, il a été préféré de conserver la consistance du modèle plutôt que d'essayer d'adapter la sémantique du `if` des langages tels que C. De ce fait, les deux branches du `if` sont mises à jour à chaque coupe et la valeur pour la coupe courante du noeud `if` est celle d'une des deux branches (selon la valeur du noeud condition). Cette description concerne la sémantique du modèle. Rien n'empêche d'implémenter une évaluation efficace du graphe qui ne calcule pas les valeurs de noeuds non utilisés (finalement par un noeud de sortie). Mais il ne faut pas s'attendre à pouvoir écrire du code qui ferait une erreur dans une des deux branches en pensant qu'il ne sera pas exécuté vu la condition. Par exemple, si prendre le premier élément (avec l'opération `head`) d'une liste vide génère une erreur, on ne peut pas écrire `(s-if (s-nul? lst) '() (s-head lst))`. Le mieux est que `head` ne fournisse pas de valeur dans le cas où `lst` est vide mais soit simplement absent.

6.7 Implémentation

6.7.1 Signaux

Structure d'un signal

Le coeur d'AEDF est écrit en Scheme, en utilisant l'implémentation appelée Chicken. L'objet principal d'AEDF est le signal. Il s'agit d'une structure (ou *record* en anglais). Cette structure comporte plusieurs champs.

Le champ `id` sert à identifier de manière unique chaque signal dans un programme dataflow. Dans l'état actuel de la bibliothèque, cela sert à générer les noms des noeuds dans un fichier source pour le logiciel Graphviz, un outil qui permet de produire des images de graphe.

Le champ `update-name` permet de connaître le type d'opération réalisée par le noeud, par exemple s'il s'agit d'un délai ou d'une addition.

Le champ `name` contient éventuellement le nom du noeud tel qu'il est donné par la macro `node`. Chaque signal contient la liste de ses parents dans le champ `parents` et la liste de ses enfants dans le champ `children`.

La procédure de mise à jour est dans le champ `update-proc`. Celle-ci est appelée lorsqu'une coupe met à jour les noeuds du graphe. Elle reçoit en argument le signal qu'elle doit mettre à jour. De cette manière, elle a accès aux valeurs des noeuds parents pour calculer la nouvelle valeur du signal (qui est également un champ). Les enfants ne sont pas véritablement nécessaires pour l'instant mais ils pourraient servir à une mise à jour selon le principe de Visual Value 3D[Nil06]. Dans VV3D, la mise à jour d'une valeur se fait en deux étapes. La première consiste à invalider les noeuds découlant d'un noeud qui vient de changer de valeur. Cette étape permet de connaître quels noeuds voient leur valeur changer. La deuxième étape consiste à calculer la valeur d'un noeud uniquement si elle est nécessaire. Ces deux étapes permettent un calcul des valeurs "à la demande", lorsqu'un noeud de sortie en a besoin.

En plus de la valeur courante, un signal possède un champ `state` qui permet de conserver un état indépendant de la valeur courante entre les appels de la procédure de mise à jour. Cet état est rarement nécessaire. Il est nécessaire pour implémenter le type de signal `s-wide`. Ce signal est un comportement qui vaut vrai pendant un certain temps à partir de l'occurrence du signal parent. Autrement, il vaut faux. Imaginons un bouton qui doit changer de couleur au moment d'un click (et reprendre immédiatement sa couleur normale). Changer de couleur dans la coupe du click

n'est pas suffisant car au moment de la coupe de rendu, le bouton aura repris sa couleur normale. `s-wide` permet de changer de couleur pendant plusieurs coupes après le click et plusieurs rendus pourront être réalisés avec la nouvelle couleur avant de revenir à l'ancienne.

Enfin, chaque signal possède la valeur de son rang dans le graphe, dans le champ `rank`. Un signal de base possède le rang 1. Un signal qui possède des parents a un rang qui vaut le maximum des rangs des parents augmenté de 1.

Création d'un signal

Créer un nouveau type de signal se fait avec la procédure `new-signal`. Le premier argument est le nom du type du signal. Le second argument est la liste des signaux dont il dépend (les signaux parents). Le troisième argument est la valeur initiale. Le quatrième argument est l'état initial. Le cinquième et dernier argument est la procédure de mise à jour. Voici comment un opérateur qui retourne un signal qui somme ses deux parents peut être défini.

```
(define (s+ s0 s1)
    (new-signal '+ (list s0 s1) 0 'no-state
                (lambda (s)
                  (signal-value-set!
                    s
                    (+ (signal-pval s 0) (signal-pval s 1))))))
```

Cette définition se lit comme suit : `s+` est une procédure qui accepte deux arguments, nommés `s0` et `s1` (les signaux parents). Cette procédure retourne un nouveau signal. A chaque mise à jour (appel du cinquième argument de `new-signal`), la valeur du signal devient la somme de la valeur du premier parent et de la valeur du second parent (obtenues par la procédure `signal-pval`). L'implémentation d'AEDF utilise une définition similaire pour d'autres opérateurs binaires. En fait, une macro a été définie pour créer un opérateur sur des signaux à partir d'un opérateur sur les valeurs (en anglais, on parle de *lifting*). Par exemple, la procédure `s+` est obtenue avec :

```
(signal-binary s+ +)
```

Une macro semblable est disponible pour convertir des opérateurs unaires.

```
(signal-unary s-cos cos)
```

Créer des signaux de sortie est un peu plus compliqué. Il faut pour cela utiliser la procédure `new-sink-kind`. Cette procédure accepte également un nom permettant d'identifier le type de noeud dont il s'agit. A la place d'une procédure de mise à jour, elle accepte une procédure appelée lorsque la coupe de sortie associée est exécutée.

6.7.2 Mise à jour du graphe

Le rang permet de connaître l'ordre de mise à jour des noeuds du graphe pendant une coupe. La mise à jour procède avec tous les noeuds d'un rang avant les noeuds du rang suivant. De cette manière, lorsque la procédure de mise à jour d'un noeud est appelée, il est garanti que les noeuds parents possèdent leur valeur pour la coupe courante et il est aussi possible de connaître l'absence d'un évènement.

Mettre à jour un noeud délai consiste simplement à copier la valeur du noeud dont il dépend. Cette copie doit être faite avant que ce dernier ne soit modifié pour la coupe courante. Pour cela, la mise à jour de noeuds délais est faite avant tous les autres. Cet effet est atteint en donnant aux noeuds délais un rang de 0. Il faut aussi prendre soin de mettre à jour les noeuds délais dans le bonne ordre car un noeud délai peut être construit à partir d'un autre noeud délai.

Certains noeuds représentent des constantes, par exemple un nombre ou une chaîne de caractères. Ces noeuds ne possèdent pas de procédure de mise à jour. Ils sont en fait des noeuds de base particuliers qui ne nécessitent pas de mise à jour.

6.7.3 Boucle principale

La boucle principale gère les évènements d'entrée et l'écoulement du temps. Le temps passe de manière discrète. Il s'agit d'un évènement comme un autre. A chaque évènement, le noeud de base qui représente cet évènement devient présent. Ensuite, une mise à jour du graphe est réalisée. Après la mise à jour, le noeud redevient absent. Il y a aussi des noeuds qui représentent des comportements. `mouse-position` en est un. Sa valeur est modifiée lorsque la souris est déplacée et puis est conservée jusqu'au prochain déplacement.

La boucle principale commence par faire avancer le temps (ici, le temps est avancé de 10 millisecondes et la boucle est répétée à 100Hz) puis effectue la gestion décrite plus haut pour tous les évènements en attente. Après, toutes les coupes de sorties sont effectuées. A chaque coupe de sortie est associé un groupe de noeuds de sorties. Pour terminer, une coupe de mise à jour est exécutée pour un évènement particulier qui marque la fin de la boucle (et la fin du rendu). Cet évènement peut servir par exemple à s'assurer qu'un rendu exploite l'apparition d'un évènement. Il suffit de

conserver un signal à vrai à partir de l'occurrence de l'évènement puis à le mettre à faux après le rendu.

6.7.4 Simplification du graphe

Idéalement, le graphe devrait être compilé. A la place de parcourir le graphe et d'appeler les procédures de mise à jour, le code compilé devrait être exécuté. La compilation n'a pas été réalisée. Par contre, une étape intermédiaire a été implémentée. Elle consiste à marquer les noeuds inutilisés. Un noeud est inutilisé lorsqu'il n'est pas un noeud de sortie et qu'il n'a pas d'enfant utilisé. La procédure de simplification commence par marquer les noeuds du dernier rang. Par définition, ces noeuds ne possèdent pas d'enfants utilisés (puisqu'ils ne possèdent pas d'enfants). Il suffit alors de vérifier si ce sont des noeuds de sorties pour savoir s'ils sont utilisés ou non. La procédure passe alors aux noeuds de l'avant-dernier rang. Ces noeuds peuvent posséder des enfants mais il a été décidé dans l'étape précédente s'ils sont utilisés ou non. Chaque rang est ainsi traité jusqu'au rang des noeuds délais.

6.7.5 Scheme et C

La boucle principale et la gestion du graphe dataflow est réalisée en Scheme. La couche de gestion des évènements et de fenêtrage est écrite avec la bibliothèque Xlib. L'affichage dans la fenêtre se fait en utilisant un contexte OpenGL (via l'extension GLX du serveur X). L'affichage 2D sous OpenGL est implémenté avec la bibliothèque de dessin vectoriel Cairo. Cairo peut rendre des images en PNG, PDF, directement dans une surface X ou en mémoire. C'est cette dernière possibilité qui est utilisée. Le tampon mémoire visé par Cairo est ensuite envoyé à OpenGL pour en faire une texture. Enfin, la texture est affichée sur un rectangle qui remplit l'écran en vue orthographique. Il s'agit d'une solution générale qui permet d'utiliser Cairo pour dessiner à n'importe quel point de l'écran. S'il faut uniquement dessiner dans une zone précise, par exemple un menu contextuel appelé par un click du bouton droit de la souris, il est préférable d'allouer un tampon de mémoire et une texture de la taille du menu plutôt que de la taille de l'écran.

Toutes les couches de bas niveau sont écrites en C (utilisation des bibliothèques Xlib, Cairo et OpenGL). Ces couches sont compilées sous forme de bibliothèques partagées. Pour chacune de ces bibliothèques, une extension pour Chicken a été écrite. Ce sont ces extensions qui sont utilisées finalement par la boucle principale. Chicken fournit une extension pour utiliser OpenGL mais elle ne comprend pas toutes les fonctionnalités offertes par OpenGL 2.1. A la place d'augmenter cette extension

ou de la refaire, une extension pour GLEW[3] a été écrite. Une bonne partie du travail a pu être automatisée en utilisant le fichier d'en-tête de GLEW et les facilités offertes par Chicken.

6.7.6 Synchronisme

Une source de motivation pour la réalisation de la bibliothèque AEDF est l'absence d'une caractéristique qui nous semble intéressante dans FrTime ou [Ell08]. Cette caractéristique est l'absence de synchronisme des occurrences de deux (ou plus) évènements qui sont dérivés d'un même parent. Prenons un exemple, illustré à la figure 6.3, et considérons un évènement qui représente l'enfoncement des touches d'un clavier. Cet évènement possède une occurrence chaque fois qu'une touche du clavier est enfoncée et prend pour valeur le caractère unicode correspondant. A partir de cet évènement, nous dérivons deux autres évènements. Le premier ne possède une occurrence que lorsque le caractère unicode est alphanumérique. Le second évènement est présent uniquement toutes les dix occurrences de son parent. Enfin, nous construisons un dernier évènement qui combine ces deux évènements dérivés. Ce dernier évènement est présent chaque fois qu'un des signaux dont il dépend est présent. Il est également présent une seule fois si ces deux signaux sont présents en même temps.

Si le signal qui représente le clavier émet pour la dixième fois et qu'il s'agit d'un caractère alphanumérique, les deux enfants seront présents et l'évènement 'combine' aussi. Dans FrTime ou dans [Ell08], la réaction à l'enfoncement de la touche sera double : une fois pour la branche gauche et une fois pour la branche droite du graphe. Dans AEDF, la présence des évènements est véritablement synchrone. Il n'y a donc qu'une seule occurrence au niveau du dernier noeud et qu'une seule réaction.

Cette caractéristique nous semble intéressante car elle simplifie le sens de certaines constructions. Par exemple, on peut utiliser des opérateurs qui créent un évènement à partir d'un comportement et inversement. L'opérateur `hold` crée un comportement qui prend pour valeur la valeur lors de la dernière occurrence de l'évènement à partir duquel il est construit. L'opérateur `changes` crée un évènement qui est présent lorsque le comportement à partir duquel il est construit change de valeur. Grâce au modèle adopté, une expression comme `(combine (changes (hold e)) e)` est équivalente à `e`.

[3] GLEW : The OpenGL Extension Wrangler Library

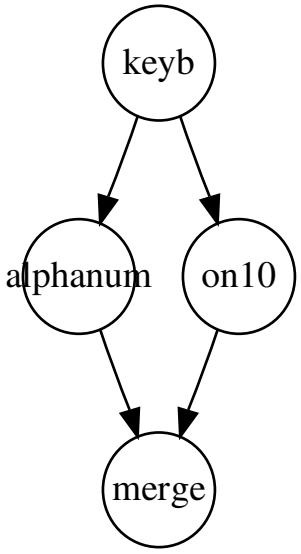


FIG. 6.3 – *Combiner deux évènements qui sont dérivés d'un seul et même parent donne un évènement dont les occurrences coïncident avec le parent.*

6.8 Application

La création de différentes interfaces faites dans la première partie de ce travail peut se faire avantageusement avec AEDF. Elle permet de se concentrer sur l'aspect dynamique de l'application. Dans cette section, un exemple complet est développé. Il s'agit d'un application simple de dessin. Quelques captures d'écran sont montrées dans la figure 6.4 et le code complet est repris à l'annexe. Le mode principal permet de tracer des lignes dans un espace 2D. Quatre autres modes sont utilisés par l'intermédiaire d'un menu. Le menu est composé de quatre zones rectangulaires placées verticalement, une pour chacun des modes.

La première zone permet de déplacer le menu en pressant dessus, en déplaçant le curseur, puis en relâchant le bouton. Les trois zones suivantes sont des boutons. Les modes associés permettent de manipuler la caméra au-dessus du dessin : translation, rotation et changement d'échelle. Pour passer dans un de ces trois modes, le bouton correspondant doit être cliqué. Cliquer à nouveau quitte le mode et l'application repasse dans le mode principal.

FIG. 6.4 – *Cette figure illustre l'application de dessin développée à la section 6.8. Quelques traits ont été tracés dans la capture d'écran de droite. La capture d'écran de gauche montre le dessin agrandi, déplacé et tourné. Le menu a également été déplacé.*

6.8.1 Nouveaux types de noeuds

La seule relative difficulté dans l'écriture de cette application est la formation et l'affichage des lignes qui forment le dessin. Comme expliqué dans la section 6.6, page 74, même avec un noeud de sortie qui représente une ligne, il n'est pas possible de permettre l'ajout non-borné de lignes supplémentaires pendant l'exécution. La solution proposée était soit de modifier AEDF pour permettre la reconfiguration du graphe à l'exécution, soit d'écrire un nouveau type de noeud de sortie. C'est cette dernière solution qui est utilisée : un noeud `s-lines` est créé pour accumuler les lignes ainsi qu'un noeud de sortie `overlay-lines` pour afficher ces lignes.

Noeud de sortie

Pour créer le nouveau type de noeud de sortie, la procédure `new-sink-kind` est utilisée.

```
(define overlay-lines
  (new-sink-kind
    *sink-group-overlay*
    'overlay-lines
    (list
      'transform (s-transform)
      'color    (s-triple 0 0 0)
      'lines (s-constant '(())))
    (lambda (re)
```

```
(let* ((data (signal-value re))
       (transform (sdata-field 'transform data))
       (color     (sdata-field 'color data))
       (lines     (sdata-field 'lines data))
       (cr (ae-cairo:gl-texture-get-cairo-context
            *cairo-overlay*)))
  (cairo-save cr)
  (cairo-translate cr (car (sdata-field 'position transform))
                      (cdr (sdata-field 'position transform)))
  (cairo-rotate    cr (sdata-field 'rotation transform))
  (cairo-scale     cr (sdata-field 'scale transform)
                      (sdata-field 'scale transform))
  (for-each (lambda (vertices) (aecr-points cr vertices))
            lines)
  (cairo-restore cr)))))
```

La valeur du noeud est utilisée dans la coupe liée au rendu de l'affichage 2D avec Cairo (symbole `*sink-group-overlay*`). Trois attributs composent ce noeud : une liste de lignes, une couleur et une transformation. La liste contient des listes de points. Chaque liste de points sera affichée en formant des segments de droite entre les points et en utilisant l'attribut couleur. La transformation permet de placer les lignes à l'écran. En modifiant la valeur de cette transformation, il sera possible de déplacer le point de vue par rapport aux lignes.

La fonction de rendu utilise les valeurs courantes des différents signaux qui forment le noeud. Elle utilise les procédures `cairo-translate`, `cairo-rotate` et `cairo-scale` pour placer les lignes en fonction de la transformation. La procédure `aecr-points` est ensuite appelée pour chaque ligne.

Noeud lignes

Les listes de points utilisés par le noeud de sortie `overlay-lines` sont fournies également par un nouveau type de noeud, créé avec le code suivant.

```
(define (s-lines s0 s1 s2)
  (new-signal 'lines (list s0 s1 s2) '(()) 'no-state
              (lambda (s)
                (let ((lines (signal-value s)))
                  (cond
                    ((signal-pval s 0)
                     (signal-value-set! s
```

```
        (cons (cons (signal-pval s 2) (car lines))
              (cdr lines))))
  ((signal-pval s 1)
   (signal-value-set! s (cons '()
                              lines)))
  ; else keeps its value
  )))))
```

La formation des lignes dépend de trois signaux `s0`, `s1` et `s2`. Le premier est un évènement qui indique qu'il faut ajouter un point au tracé courant. Le second est aussi un évènement. Il indique que la ligne courante est terminée et qu'une nouvelle commence. Le troisième signal doit valoir la position du point à ajouter lorsqu'il y a une occurrence du premier évènement. Dans notre cas, il s'agira de la position de la souris. La procédure de mise à jour découle de cette description. A chaque appel, si aucun des deux évènements n'est présent, la valeur du signal reste la même. Si le premier signal est présent, la valeur du troisième parent est ajoutée à la dernière ligne ajoutée. Si c'est le second signal qui est présent, alors une nouvelle ligne, initialement vide, est ajoutée à la liste des lignes. L'état initial du signal est une liste qui contient une seule ligne vide.

6.8.2 Positionnement du menu

La première zone du menu sert à le déplacer. Le changement de position se fait lorsque l'on presse le bouton gauche de la souris au-dessus de cette zone puis déplace le curseur. Reformulé sous une forme plus proche d'AEDF, la position `pos0` de la première zone est soit sa position à la coupe précédente, soit une position qui dépend de sa position précédente. La position précédente `pos0-d` est obtenue avec l'opération `s-delay`. Comme `pos0` dépend de son passé, il faut l'initialiser explicitement.

```
(node pos0-d (s-delay pos0))
(node pos0 (s-if condition
                 (... pos0-d ...)
                 pos0-d))
(initial-value pos0 (cons 10 10)
```

La condition pour le déplacement est appelée `manip0`. `manip0` doit être vrai lorsque l'on a pressé dans la zone et devenir faux lorsque l'on relâche le bouton. La procédure `toggle2` permet de créer le comportement voulu. `at-press0` (défini plus loin) est

un évènement qui a lieu lorsque l'on presse dans la zone et `at-lmb-release` est un évènement qui a lieu lorsque le bouton gauche de la souris est relâché.

```
(node manip0 (s-toggle2 at-press0 at-lmb-release))
```

Les cycles dans le graphe ne sont permis que s'ils ne sont pas instantanés (ils doivent comporter un délai). Comme `at-press0` est défini en fonction de la position de la zone et que cette position est définie en fonction de `manip0`, nous venons de créer une boucle instantanée. Nous modifions donc la condition pour qu'elle soit `(s-delay manip0)`. Pour compléter la définition de `pos0`, il reste à mentionner qu'il existe deux signaux de base `mouse-dx` et `mouse-dy` qui ont pour valeur la différence entre la position actuelle du curseur et la position précédente. Ajouter cette différence à la position précédente de la zone permet de lui faire suivre le curseur. La définition complète de `pos0` devient :

```
(node pos0 (s-if (s-and manip0-d at-mouse-move)
                 (s-cons (s+ (s-car pos0-d) mouse-dx)
                         (s+ (s-cdr pos0-d) mouse-dy))
                 pos0-d))
```

6.8.3 Création des zones du menu

Pour donner corps à la zone, un signal de sortie `overlay-rectangle` est créé. Il sera affiché via Cairo.

```
(node rect0 (overlay-rectangle 'position pos0))
```

Il est maintenant possible de définir `at-press0` :

```
(node at-press0
  (s-and at-lmb-press
         (s-in-rectangle? mouse-position rect0)))
```

Les trois zones restantes sont plus simples à positionner : la position de chacune est celle de la première avec un décalage sur l'axe vertical. Comme pour la première zone, un rectangle et un signal permettant de connaître lorsqu'ils sont pressés leurs sont associés.

```
(node pos1 (s-cons (s-car pos0) (s+ (s-cdr pos0) 30)))
(node pos2 (s-cons (s-car pos0) (s+ (s-cdr pos0) 60)))
(node pos3 (s-cons (s-car pos0) (s+ (s-cdr pos0) 90)))

(node rect1 (overlay-rectangle 'position pos1))
(node rect2 (overlay-rectangle 'position pos2))
(node rect3 (overlay-rectangle 'position pos3))

(node at-press1
  (s-and at-lmb-press
         (s-in-rectangle? mouse-position rect3)))
(node at-press2
  (s-and at-lmb-press
         (s-in-rectangle? mouse-position rect2)))
(node at-press3
  (s-and at-lmb-press
         (s-in-rectangle? mouse-position rect3)))
```

6.8.4 Association des zones aux opérations de navigation

A chacun des trois boutons est associé un signal booléen (nommé `active`) indiquant si l'opération associée (translation, rotation ou mise à l'échelle) est utilisée. Ce signal démarre initialement avec la valeur faux. Il devient vrai lorsque le bouton est pressé et redevient faux lorsque l'on presse à nouveau dessus ou, pour assurer qu'un seul mode ne soit actif à la fois, lorsque l'on presse sur un autre bouton (mais pas la première zone qui permet de déplacer le menu). Pour détecter si un des trois boutons est pressé, il suffit de créer un signal avec l'opérateur `s-or`.

```
(node at-press-op
  (s-or (s-or at-press1 at-press2)
        at-press3))
```

Le comportement décrit peut être accompli grâce à l'opérateur `toggle`. Cet opérateur prend un évènement et retourne un signal booléen qui passe d'une valeur à l'autre à chaque occurrence de l'évènement. Sa valeur initiale est faux. Pour connaître si le bouton était déjà actif, un délai est utilisé.

```
(node active1-d (s-delay active1))
(node active1 (s-toggle (s-or at-press1
                              (s-and active1-d at-press-op))))
```

```
(node active2-d (s-delay active2))
(node active2 (s-toggle (s-or at-press2
                              (s-and active2-d at-press-op))))
(node active3-d (s-delay active3))
(node active3 (s-toggle (s-or at-press3
                              (s-and active3-d at-press-op))))
```

Jusqu'à présent, nous avons affiché les zones avec des rectangles. Ces rectangles nous ont également permis de définir quelques signaux. Nous allons conserver ces rectangles pour l'aspect logique mais les remplacer par des labels pour l'affichage. De plus, nous allons pouvoir utiliser les signaux définis pour modifier la couleur des labels. Cette écriture en deux couches (les rectangles pour la relation entre zones et curseur d'une part et les labels pour l'affichage d'autre part) évite d'utiliser des délais (le rectangle dépend de sa couleur et la couleur dépend du curseur et du rectangle). Pour que les rectangles ne soient pas visibles au rendu, il suffit de mettre à faux l'attribut `'render`.

```
(node rect0 (overlay-rectangle 'render #f 'position pos0))
(node rect1 (overlay-rectangle 'render #f 'position pos1))
(node rect2 (overlay-rectangle 'render #f 'position pos2))
(node rect3 (overlay-rectangle 'render #f 'position pos3))
```

Les labels sont placés au-dessus des rectangles. Les couleurs dépendent de `manip0` pour le premier et des `active` pour les trois autres.

```
(node label0
  (label
    'text (s-string "menu")
    'position pos0
    'color (s-if manip0
                 (s-triple 0 1 1)
                 (s-triple 0 0 0))))
(node label1
  (label
    'text (s-string "translate")
    'position pos1
    'color (s-if active1
                 (s-triple 0 1 1)
                 (s-triple 0 0 0))))
(node label2
```

```
  (label
    'text (s-string "rotate")
    'position pos2
    'color (s-if active2
                 (s-triple 0 1 1)
                 (s-triple 0 0 0))))
(node label3
  (label
    'text (s-string "scale")
    'position pos3
    'color (s-if active3
                 (s-triple 0 1 1)
                 (s-triple 0 0 0))))
```

6.8.5 Transformation de navigation

Pour terminer l'application, il reste à associer une transformation au dessin. Les valeurs qui composent la transformation sont contrôlées par les mouvements de la souris et les valeurs des signaux liés aux boutons. Ces différentes valeurs sont définies de manière similaire à la position de la première zone (`pos0`). Pour cette définition, nous avons utilisé un délai (`pos0-d`). Ici, nous utilisons un nouvel opérateur `acc`. Le signal résultant démarre avec une certaine valeur (le deuxième argument) et utilise le premier argument pour accumuler les valeurs du quatrième argument chaque fois que le troisième survient. Par exemple, la rotation commence avec un angle de 0° et additionne la valeur de `mouse-dy` à chaque mouvement de souris (à condition que le bouton gauche soit pressé et que le troisième bouton soit actif).

```
(node scene-translation
  (s-acc add-pair
         (cons 0 0)
         (s-and active1
                (s-and lmb-press at-mouse-move))
         mouse-delta))
(node scene-rotation
  (s-acc +
         0
         (s-and active2
                (s-and lmb-press at-mouse-move))
         (s-radians mouse-dy)))
(node scene-scale
  (s-acc +
```

```
          1
          (s-and active3
                 (s-and lmb-press at-mouse-move))
          (s/ mouse-dy 60.0)))
(node scene-transform
  (s-transform 'position scene-translation
               'rotation scene-rotation
               'scale scene-scale))
```

6.8.6 Clôture du programme

Quelques modifications au programme de dessin doivent être apportées. La création des lignes doit être modifiée pour que de nouveaux points ne soient ajoutés que si aucun bouton n'est actif. L'affichage doit dépendre de la transformation que nous venons de définir. La transformation est également utilisée pour modifier les coordonnées de la position du curseur avant de l'ajouter à la ligne en cours (`imulxp` multiplie l'inverse (le caractère 'i') d'une transformation (le 'x') avec un point (le 'p')).

```
(node lines (s-lines (s-and (s-not any-active)
                            (s-and lmb-press
                                   at-mouse-move))
                     (s-and (s-not any-active)
                            at-lmb-release)
                     (s-imulxp scene-transform mouse-position)))
(node o-lines (overlay-lines 'transform scene-transform
                             'lines lines))
```

6.9 Extensions

Bien que la bibliothèque AEDF se révèle utile dans son état actuel (comme l'a montré le développement d'une petite application à la section précédente), il est possible de l'améliorer considérablement. Cette section présente trois extensions avantageuses importantes.

6.9.1 Threads

Comme l'a montré la section sur la comparaison avec FairThread, AEDF est proche d'une librairie ou d'un langage pour la programmation multi-threads. Il serait probablement intéressant d'ajouter à la bibliothèque la possibilité d'écrire des portions de graphes sous forme de code séquentiel qui attend et émet des évènements (les threads). En pratique, ces threads sont des machines à états. Pour faciliter l'écriture de telles machines (sauvegarder l'état courant atteint au cours d'une coupe pour reprendre à cet endroit à la coupe suivante), une solution élégante peut être basée sur les continuations dans le cas de Scheme.

6.9.2 Reconfiguration dynamique

Une extension importante concerne la reconfiguration dynamique du graphe dataflow. La possibilité la plus évidente qui évite de toucher au graphe durant une mise à jour est d'utiliser des évènements qui déclenchent la modification du graphe dans une coupe similaire à une coupe de sorties (assimilable à une partie du programme qui permet les effets de bord). Essentiellement, le principe est d'ajouter une nouvelle coupe dans la boucle principale. Cette coupe est associée à de nouveaux types de noeuds. Tout comme il y a des noeuds de sorties, nous aurions alors des noeuds de reconfigurations. Ces noeuds permettraient d'ajouter, de supprimer ou de déplacer des noeuds existants. Une telle coupe, comme une coupe de sortie, utilise l'état du graphe sans le mettre à jour. A l'usage, ces noeuds dépendraient probablement plus d'évènements que de comportements car la modification du graphe représente un changement important dans l'état du programme ; ce changement aurait sans doute lieu à des moments bien précis dans le temps. De plus, maintenir un graphe dynamique est plus lourd à l'exécution qu'un graphe statique qui peut être réécrit automatiquement sous une forme plus efficace.

6.9.3 Compilation

Le graphe dataflow que reçoit `aedf:run` fait immanquablement penser à un arbre syntaxique abstrait. Par exemple, l'expression `(node a (s+ b c))` est représentée dans la section 6.2. Cette expression est similaire dans un langage impératif à l'expression `a = b + c`. De même, le mécanisme de mise à jour utilisé à chaque coupe consiste à calculer d'abord les valeurs de `b` et `c` puis à calculer celle de `a`. Cette mise à jour et le code résultant de la compilation de l'expression impérative ont un fonctionnement identique. Cette simple constatation appelle donc à compiler le graphe. Après compilation, mettre à jour le graphe ne consiste plus à parcourir les noeuds et à exécuter leur procédure de mise à jour en leur passant les parents du noeud,

mais simplement à exécuter le code compilé.

Le graphe pourrait être analysé plus finement pour réduire au maximum les calculs nécessaires. Les portions redondantes doivent être éliminées. Les mises à jour ne doivent concerner que les noeuds affectés (ceux qui dépendent des noeuds de base modifiés pendant la coupe courante). Une évaluation paresseuse pourrait être utilisée pour ne mettre à jour les noeuds que si leur valeur est nécessaire pour une coupe de sorties (s'il n'est pas possible de le décider statiquement).

La compilation du graphe pourrait être étendue à la réalisation d'une boucle principale. Elle serait synthétisée en fonction du graphe donné et ne contiendrait ensuite que les coupes nécessaires (celles associées aux noeuds de base qui n'ont pas été jugés inutiles dans l'étape de simplification).

Conclusion et perspectives

Dans ce mémoire, nous avons eu pour objectif de réaliser des prototypes de widgets pour la navigation dans des scènes virtuelles 2D et 3D pour les applications EsQUIsE et SketSha développées au Lucid Group, le laboratoire de recherche du département ArGEnCo (Architecture, Géologie, Environnement & Constructions) de la Faculté des Sciences Appliquées de l'Université de Liège.

Ces applications ont été créées pour fonctionner avec le Bureau Virtuel et sont utilisées par des architectes lors de la phase de conception. Le Bureau Virtuel et EsQUIsE doivent résoudre des difficultés spécifiques et nouvelles que les autres solutions de CAO ne maîtrisent pas. Ils proposent une interface intuitive pour l'architecte en simulant un environnement de travail familier : le papier et le crayon. Ils complètent le processus d'exploration graphique mené par le concepteur en lui fournissant des évaluations de performances issues des techniques d'ingénierie du bâtiment ainsi que la visualisation d'un modèle 3D. Les évaluations et le modèle 3D sont générés directement à partir des croquis du concepteur.

La visualisation de modèles en trois dimensions de bâtiments demande de pouvoir contrôler le point de vue de manière intuitive. Les techniques d'interactions employées doivent être adaptées à la taille du Bureau Virtuel et se satisfaire du stylet comme périphérique d'entrée.

Nous avons développé des prototypes de widgets afin de répondre à ces exigences par l'usage de techniques d'interactions puisées dans la littérature scientifique, telles que les *marking menus*, les interfaces à base de croisements ou le *Curve Dial*, pour contrôler les opérations de navigation.

Les widgets ont pu être expérimentés par des membres de l'équipe ainsi que par des étudiants en architecture. Les widgets utilisés par les étudiants offrent une interface originale dédiée au problème de la navigation. Il permet de contrôler aisément le placement du centre de rotation et de mise à l'échelle. Son exploitation sur la grande surface du Bureau Virtuel avec un stylet partageable par un groupe de personnes est facilitée par un mécanisme simple d'appel. Des modifications ont été apportées au

logiciel SketSha pour permettre une évaluation systématique ultérieure des widgets. Des widgets pour la navigation dans des scènes 3D ont été réalisés également.

Nous avons souhaité prolonger cette recherche par la création d'une bibliothèque de programmation. Cette bibliothèque présente un intérêt particulier à permettre l'écriture rapide d'applications interactives. Nous l'employons à la création d'une application simple de dessin qui utilise une variante des widgets de navigation. Cette bibliothèque exploite le principe d'une évaluation dataflow. Les programmes écrits avec cette bibliothèque sont courts et simple à comprendre. Ils expriment de façon concise le comportement attendu de l'application (ou du widget). Cette bibliothèque est donc un terrain idéal pour concevoir et expérimenter à l'avenir la création de nouveaux types de widgets.

Dans le dernier chapitre, nous dégageons trois extensions pour la bibliothèque dataflow qui nous ont paru importantes à implémenter : le multi-threading, la reconfiguration dynamique et la compilation du graphe dataflow.

Bibliographie

[AG05] Georg Apitz and François Guimbretière. Crossy : a crossing-based drawing application. In *SIGGRAPH '05 : ACM SIGGRAPH 2005 Papers*, pages 930–930, New York, NY, USA, 2005. ACM.

[AZ02] Johnny Accot and Shumin Zhai. More than dotting the i's — foundations for crossing-based interfaces. In *CHI '02 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 73–80, New York, NY, USA, 2002. ACM.

[Ber00] Gérard Berry. The foundations of esterel. In *Proof, language, and interaction : essays in honour of Robin Milner*, pages 425–454, Cambridge, MA, USA, 2000. MIT Press.

[BSF+95] Eric A. Bier, Maureen C. Stone, Ken Fishkin, William A. S. Buxton, and Thomas Baudel. A taxonomy of see-through tools. In *Human-computer interaction : toward the year 2000*, pages 517–523. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[CK06] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, pages 294–308, 2006.

[Ell08] Conal Elliott. Simply efficient functional reactivity. *Unpublished draft*, 2008.

[FCF+02] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme : A programming environment for scheme. *Journal of Functional Programming*, 12 :369–388, 2002.

[FKP+03] George Fitzmaurice, Azam Khan, Robert Pieké, Bill Buxton, and Gordon Kurtenbach. Tracking menus. In *UIST '03 : Proceedings of the 16th annual ACM symposium on User interface software and technology*, pages 71–79, New York, NY, USA, 2003. ACM.

[FVB06] Clifton Forlines, Daniel Vogel, and Ravin Balakrishnan. Hybridpointing : fluid switching between absolute and relative pointing with a

direct input device. In *UIST '06 : Proceedings of the 19th annual ACM symposium on User interface software and technology*, pages 211–220, New York, NY, USA, 2006. ACM.

[FvDFH96] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer graphics (2nd ed. in C) : principles and practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[GHB+06] Tovi Grossman, Ken Hinckley, Patrick Baudisch, Maneesh Agrawala, and Ravin Balakrishnan. Hover widgets : using the tracking state to extend the capabilities of pen-operated devices. In *CHI '06 : Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 861–870, New York, NY, USA, 2006. ACM.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

[GW00] François Guimbretière and Terry Winograd. Flowmenu : combining command, text, and data entry. In *UIST '00 : Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 213–216, New York, NY, USA, 2000. ACM.

[HCNP03] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming, 4th International School, volume 2638 of LNCS*, pages 159–187. Springer-Verlag, 2003.

[Hul90] Jeff Hultquist. A virtual trackball. In *Graphics gems*, pages 462–463. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[JL04] Roland Juchmes and Pierre Leclercq. Esquise-sma : un système multi-agents pour l'interprétation d'esquisses architecturales. In *IHM*, pages 179–180, 2004.

[KB93] Gordon Kurtenbach and William Buxton. The limits of expert performance using hierarchic marking menus. In *CHI '93 : Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, pages 482–487, New York, NY, USA, 1993. ACM.

[KB94] Gordon Kurtenbach and William Buxton. User learning and performance with marking menus. In *CHI '94 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 258–264, New York, NY, USA, 1994. ACM.

[KCe+98] Richard Kelsey, William Clinger, Jonathan Rees (editors, H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised 5 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 33 :26–76, 1998.

[KKS+05] Azam Khan, Ben Komalo, Jos Stam, George Fitzmaurice, and Gordon Kurtenbach. Hovercam : interactive 3d navigation for proximal object inspection. In *I3D '05 : Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 73–80, New York, NY, USA, 2005. ACM.

[Kur93] Gordon Paul Kurtenbach. *The design and evaluation of marking menus*. PhD thesis, Toronto, Ont., Canada, Canada, 1993.

[LGC99] Mark A. Livingston, Arthur Gregory, and Bruce Culbertson. Six degree of freedom control with a two-dimensional input device : Intuitive controls and simple implementations. 1999.

[Nil06] Anders Nilsson. Visual value 3d, efficient event propagation in scene graphs. Master's thesis, Lund, Sweden, 2006.

[SBL05] Stéphane Safin, Christelle Boulanger, and Pierre Leclercq. Premières évaluations d'un bureau virtuel pour un processus de conception augmenté. In *IHM*, volume 264 of *ACM International Conference Proceeding Series*, pages 107–114. ACM, 2005.

[Sho94] Ken Shoemake. Arcball rotation control. In *Graphics gems IV*, pages 175–192. Academic Press Professional, Inc., San Diego, CA, USA, 1994.

[Smcs04] G. M. Smith and m. c. schraefel. The radial scroll tool : scrolling support for stylus- or touch-based document navigation. In *UIST '04 : Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 53–56, New York, NY, USA, 2004. ACM.

[SmcsB05] Grham Smith, m. c. schraefel, and Patrick Baudisch. Curve dial : eyes-free parameter entry for guis. In *CHI '05 : CHI '05 extended abstracts on Human factors in computing systems*, pages 1146–1147, New York, NY, USA, 2005. ACM.

[SRD00] Henry Sowizral, Kevin Rushforth, and Michael Deering. *The Java 3d API Specification with Cdrom*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[SWND05] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.

[WO90] Colin Ware and Steven Osborne. Exploration and virtual camera control in virtual three dimensional environments. In *SI3D '90 : Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 175–183, New York, NY, USA, 1990. ACM.

[ZAH06] Shengdong Zhao, Maneesh Agrawala, and Ken Hinckley. Zone and polygon menus : using relative position to increase the breadth of multi-stroke marking menus. In *CHI '06 : Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 1077–1086, New York, NY, USA, 2006. ACM.

[ZB04] Shengdong Zhao and Ravin Balakrishnan. Simple vs. compound mark hierarchical marking menus. In *UIST '04 : Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 33–42, New York, NY, USA, 2004. ACM.

[ZF99] Robert Zeleznik and Andrew Forsberg. Unicam—2d gestural camera controls for 3d environments. In *I3D '99 : Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 169–173, New York, NY, USA, 1999. ACM.

Annexe

Dans la section 6.8, page 81, une application de dessin simple a été écrite. Le code complet de cette application est repris ici.

```
; 4 buttons menu, buttons numbered from 0 to 3.
; 0 lets the user move the menu around.
; 1 lets the user activate the 'translate' operation
; 2 lets the user activate the 'rotate' operation
; 3 lets the user activate the 'scale' operation
;

; is manip0 #t at the previous slice ?
(node manip0-d (s-delay manip0))

; position of the first button. can be done with s-acc
; instead of s-delay.
(node pos0-d (s-delay pos0))
(node pos0 (s-if (s-and manip0-d at-mouse-move)
                 (s-cons (s+ (s-car pos0-d) mouse-dx)
                         (s+ (s-cdr pos0-d) mouse-dy))
                 pos0-d))
; because there is a cycle, pos0's value can't be initialized
; automatically and thus it should be done explicitely.
(signal-value-set! pos0 (cons 10 10))

; positions of the operation buttons.
(node pos1 (s-cons (s-car pos0) (s+ (s-cdr pos0) 30)))
(node pos2 (s-cons (s-car pos0) (s+ (s-cdr pos0) 60)))
(node pos3 (s-cons (s-car pos0) (s+ (s-cdr pos0) 90)))

; rectangles of the buttons for 'picking'.
(node rect0 (overlay-rectangle 'render #f 'position pos0))
```

```
(node rect1 (overlay-rectangle 'render #f 'position pos1))
(node rect2 (overlay-rectangle 'render #f 'position pos2))
(node rect3 (overlay-rectangle 'render #f 'position pos3))

; at-press* events -- when the user presses in a button
; (more specifically, its rectangle).
(node at-press0
  (s-and at-lmb-press
         (s-in-rectangle? mouse-position rect0)))
(node at-press1
  (s-and at-lmb-press
         (s-in-rectangle? mouse-position rect1)))
(node at-press2
  (s-and at-lmb-press
         (s-in-rectangle? mouse-position rect2)))
(node at-press3
  (s-and at-lmb-press
         (s-in-rectangle? mouse-position rect3)))

; boolean signal : #t from a press in rect0 then #f when
; release anywhere.
(node manip0 (s-toggle2 at-press0 at-lmb-release))

; at-press* event -- when the user presses any
; operation-related button.
(node at-press-op
  (s-or (s-or at-press1 at-press2)
        at-press3))

; is the button activated ?
(node active1-d (s-delay active1))
(node active1 (s-toggle (s-or at-press1
                              (s-and active1-d at-press-op))))
(node active2-d (s-delay active2))
(node active2 (s-toggle (s-or at-press2
                              (s-and active2-d at-press-op))))
(node active3-d (s-delay active3))
(node active3 (s-toggle (s-or at-press3
                              (s-and active3-d at-press-op))))

(node any-active (s-or manip0
                       (s-or active1
```

```
                              (s-or active2
                                    active3))))

; labels, i.e. button graphical representations (displayed
; at the same position than the 'picking' rectangles).
(node label0
  (label
    'text (s-string "menu")
    'position pos0
    'color (s-if manip0
                 (s-triple 0 1 1)
                 (s-triple 0 0 0))))

(node label1
  (label
    'text (s-string "translate")
    'position pos1
    'color (s-if active1
                 (s-triple 0 1 1)
                 (s-triple 0 0 0))))

(node label2
  (label
    'text (s-string "rotate")
    'position pos2
    'color (s-if active2
                 (s-triple 0 1 1)
                 (s-triple 0 0 0))))

(node label3
  (label
    'text (s-string "scale")
    'position pos3
    'color (s-if active3
                 (s-triple 0 1 1)
                 (s-triple 0 0 0))))

(node scene-translation
  (s-acc add-pair
         (cons 0 0)
         (s-and active1
                (s-and lmb-press at-mouse-move))
```

```
         mouse-delta))

(node scene-rotation
  (s-acc +
         0
         (s-and active2
                (s-and lmb-press at-mouse-move))
         (s-radians mouse-dy)))

(node scene-scale
  (s-acc +
         1
         (s-and active3
                (s-and lmb-press at-mouse-move))
         (s/ mouse-dy 60.0)))

(node scene-transform
  (s-transform 'position scene-translation
               'rotation scene-rotation
               'scale scene-scale))

(node lines (s-lines (s-and (s-not any-active)
                            (s-and lmb-press
                                   at-mouse-move))
                     (s-and (s-not any-active)
                            at-lmb-release)
                     (s-imulxp scene-transform mouse-position)))
(node o-lines (overlay-lines 'transform scene-transform
                             'lines lines))
```